CHAPTER 13

Machine Learning

What Is Machine Learning

The goal of *machine learning* (ML)* is to turn data into information. After learning from a collection of data, we want a machine to be able to answer questions about the data: What other data is most similar to this data? Is there a car in the image? What ad will the user respond to? There is often a cost component, so this question could become: "Of the products that we make the most money from, which one will the user most likely buy if we show them an ad for it?" Machine learning turns data into information by extracting rules or patterns from that data.

Training and Test Set

Machine learning works on data such as temperature values, stock prices, color intensities, and so on. The data is often preprocessed into *features*. We might, for example, take a database of 10,000 face images, run an edge detector on the faces, and then collect features such as edge direction, edge strength, and offset from face center for each face. We might obtain 500 such values per face or a feature vector of 500 entries. We could then use machine learning techniques to construct some kind of model from this collected data. If we only want to see how faces fall into different groups (wide, narrow, etc.), then a *clustering* algorithm would be the appropriate choice. If we want to learn to predict the age of a person from (say) the pattern of edges detected on his or her face, then a *classifier* algorithm would be appropriate. To meet our goals, machine learning algorithms analyze our collected features and adjust weights, thresholds, and other parameters to maximize performance according to those goals. This process of parameter adjustment to meet a goal is what we mean by the term *learning*.

* Machine learning is a vast topic. OpenCV deals mostly with statistical machine learning rather than things that go under the name "Bayesian networks", "Markov random fields", or "graphical models". Some good texts in machine learning are by Hastie, Tibshirani, and Friedman [Hastie01], Duda and Hart [Duda73], Duda, Hart, and Stork [Duda00], and Bishop [Bishop07]. For discussions on how to parallelize machine learning, see Ranger et al. [Ranger07] and Chu et al. [Chu07].

It is always important to know how well machine learning methods are working, and this can be a subtle task. Traditionally, one breaks up the original data set into a large training set (perhaps 9,000 faces, in our example) and a smaller test set (the remaining 1,000 faces). We can then run our classifier over the training set to learn our age prediction model given the data feature vectors. When we are done, we can test the age prediction classifier on the remaining images in the test set.

The test set is not used in training, and we do not let the classifier "see" the test set age labels. We run the classifier over each of the 1,000 faces in the test set of data and record how well the ages it predicts from the feature vector match the actual ages. If the classifier does poorly, we might try adding new features to our data or consider a different type of classifier. We'll see in this chapter that there are many kinds of classifiers and many algorithms for training them.

If the classifier does well, we now have a potentially valuable model that we can deploy on data in the real world. Perhaps this system will be used to set the behavior of a video game based on age. As the person prepares to play, his or her face will be processed into 500 (edge direction, edge strength, offset from face center) features. This data will be passed to the classifier; the age it returns will set the game play behavior accordingly. After it has been deployed, the classifier sees faces that it never saw before and makes decisions according to what it learned on the training set.

Finally, when developing a classification system, we often use a validation data set. Sometimes, testing the whole system at the end is too big a step to take. We often want to tweak parameters along the way before submitting our classifier to final testing. We can do this by breaking the original 10,000-face data set into three parts: a training set of 8,000 faces, a validation set of 1,000 faces, and a test set of 1,000 faces. Now, while we're running through the training data set, we can "sneak" pretests on the validation data to see how we are doing. Only when we are satisfied with our performance on the validation set do we run the classifier on the test set for final judgment.

Supervised and Unsupervised Data

Data sometimes has no labels; we might just want to see what kinds of groups the faces settle into based on edge information. Sometimes the data has labels, such as age. What this means is that machine learning data may be *supervised* (i.e., may utilize a teaching "signal" or "label" that goes with the data feature vectors). If the data vectors are unlabeled then the machine learning is *unsupervised*.

Supervised learning can be *categorical*, such as learning to associate a name to a face, or the data can have *numeric* or *ordered* labels, such as age. When the data has names (categories) as labels, we say we are doing *classification*. When the data is numeric, we say we are doing *regression*: trying to fit a numeric output given some categorical or numeric input data.

Supervised learning also comes in shades of gray: It can involve one-to-one pairing of labels with data vectors or it may consist of *deferred learning* (sometimes called

reinforcement learning). In reinforcement learning, the data label (also called the *reward* or *punishment*) can come long after the individual data vectors were observed. When a mouse is running down a maze to find food, the mouse may experience a series of turns before it finally finds the food, its reward. That reward must somehow cast its influence back on all the sights and actions that the mouse took before finding the food. Reinforcement learning works the same way: the system receives a delayed signal (a reward or a punishment) and tries to infer a policy for future runs (a way of making decisions; e.g., which way to go at each step through the maze). Supervised learning can also have partial labeling, where some labels are missing (this is also called *semisupervised learning*), or noisy labels, where some labels are just wrong. Most ML algorithms handle only one or two of the situations just described. For example, the ML algorithms might handle classification but not regression; the algorithm might be able to do semisupervised learning but not reinforcement learning; the algorithm might be able to deal with numeric but not categorical data; and so on.

In contrast, often we don't have labels for our data and are interested in seeing whether the data falls naturally into groups. The algorithms for such unsupervised learning are called *clustering algorithms*. In this situation, the goal is to group unlabeled data vectors that are "close" (in some predetermined or possibly even some learned sense). We might just want to see how faces are distributed: Do they form clumps of thin, wide, long, or short faces? If we're looking at cancer data, do some cancers cluster into groups having different chemical signals? Unsupervised clustered data is also often used to form a feature vector for a higher-level supervised classifier. We might first cluster faces into face types (wide, narrow, long, short) and then use that as an input, perhaps with other data such as average vocal frequency, to predict the gender of a person.

These two common machine learning tasks, classification and clustering, overlap with two of the most common tasks in computer vision: recognition and segmentation. This is sometimes referred to as "the what" and "the where". That is, we often want our computer to name the object in an image (recognition, or "what") and also to say where the object appears (segmentation, or "where"). Because computer vision makes such heavy use of machine learning, OpenCV includes many powerful machine learning algorithms in the ML library, located in the *…/ opencv/ml* directory.

The OpenCV machine learning code is general. That is, although it is highly useful for vision tasks, the code itself is not specific to vision. One could learn, say, genomic sequences using the appropriate routines. Of course, our concern here is mostly with object recognition given feature vectors derived from images.

Generative and Discriminative Models

Many algorithms have been devised to perform learning and clustering. OpenCV supports some of the most useful currently available statistical approaches to machine learning. Probabilistic approaches to machine learning, such as Bayesian networks

or graphical models, are less well supported in OpenCV, partly because they are newer and still under active development. OpenCV tends to support *discriminative algorithms*, which give us the probability of the label given the data ($P(L \mid D)$), rather than *generative algorithms*, which give the distribution of the data given the label ($P(D \mid L)$). Although the distinction is not always clear, discriminative models are good for yielding predictions given the data while generative models are good for giving you more powerful representations of the data or for conditionally synthesizing new data (think of "imagining" an elephant; you'd be generating data given a condition "elephant").

It is often easier to interpret a generative model because it models (correctly or incorrectly) the cause of the data. Discriminative learning often comes down to making a decision based on some threshold that may seem arbitrary. For example, suppose a patch of road is identified in a scene partly because its color "red" is less than 125. But does this mean that red = 126 is definitely not road? Such issues can be hard to interpret. With generative models you are usually dealing with conditional distributions of data given the categories, so you can develop a feel for what it means to be "close" to the resulting distribution.

OpenCV ML Algorithms

The machine learning algorithms included in OpenCV are given in Table 13-1. All algorithms are in the *ML* library with the exception of Mahalanobis and K-means, which are in *CVCORE*, and face detection, which is in *CV*.

Table 13-1. Machine learning algorithms supported in OpenCV, original references to the algorithms are provided after the descriptions

Algorithm	Comment
Mahalanobis	A distance measure that accounts for the "stretchiness" of the data space by dividing out the covariance of the data. If the covariance is the identity matrix (identical variance), then this measure is identical to the Euclidean distance measure [Mahalanobis36].
K-means	An unsupervised clustering algorithm that represents a distribution of data using *K* centers, where *K* is chosen by the user. The difference between this algorithm and expectation maximization is that here the centers are not Gaussian and the resulting clusters look more like soap bubbles, since centers (in effect) compete to "own" the closest data points. These cluster regions are often used as sparse histogram bins to represent the data. Invented by Steinhaus [Steinhaus56], as used by Lloyd [Lloyd57].
Normal/Naïve Bayes classifier	A generative classifier in which features are assumed to be Gaussian distributed and statistically independent from each other, a strong assumption that is generally not true. For this reason, it's often called a "naïve Bayes" classifier. However, this method often works surprisingly well. Original mention [Maron61; Minsky61].
Decision trees	A discriminative classifier. The tree finds one data feature and a threshold at the current node that best divides the data into separate classes. The data is split and we recursively repeat the procedure down the left and right branches of the tree. Though not often the top performer, it's often the first thing you should try because it is fast and has high functionality [Breiman84].

Table 13-1. Machine learning algorithms supported in OpenCV, original references to the algorithms are provided after the descriptions (continued)

Algorithm	Comment
Boosting	A discriminative group of classifiers. The overall classification decision is made from the combined weighted classification decisions of the group of classifiers. In training, we learn the group of classifiers one at a time. Each classifier in the group is a "weak" classifier (only just above chance performance). These weak classifiers are typically composed of single-variable decision trees called "stumps". In training, the decision stump learns its classification decisions from the data and also learns a weight for its "vote" from its accuracy on the data. Between training each classifier one by one, the data points are re-weighted so that more attention is paid to data points where errors were made. This process continues until the total error over the data set, arising from the combined weighted vote of the decision trees, falls below a set threshold. This algorithm is often effective when a large amount of training data is available [Freund97].
Random trees	A discriminative forest of many decision trees, each built down to a large or maximal splitting depth. During learning, each node of each tree is allowed to choose splitting variables only from a random subset of the data features. This helps ensure that each tree becomes a statistically independent decision maker. In run mode, each tree gets an unweighted vote. This algorithm is often very effective and can also perform regression by averaging the output numbers from each tree [Ho95]; implemented: [Breiman01].
Face detector / Haar classifier	An object detection application based on a clever use of boosting. The OpenCV distribution comes with a trained frontal face detector that works remarkably well. You may train the algorithm on other objects with the software provided. It works well for rigid objects and characteristic views [Viola04].
Expectation maximization (EM)	A generative unsupervised algorithm that is used for clustering. It will fit *N* multidimensional Gaussians to the data, where *N* is chosen by the user. This can be an effective way to represent a more complex distribution with only a few parameters (means and variances). Often used in segmentation. Compare with K-means listed previously [Dempster77].
K-nearest neighbors	The simplest possible discriminative classifier. Training data are simply stored with labels. Thereafter, a test data point is classified according to the majority vote of its K nearest other data points (in a Euclidean sense of nearness). This is probably the simplest thing you can do. It is often effective but it is slow and requires lots of memory [Fix51].
Neural networks / Multilayer perceptron (MLP)	A discriminative algorithm that (almost always) has "hidden units" between output and input nodes to better represent the input signal. It can be slow to train but is very fast to run. Still the top performer for things like letter recognition [Werbos74; Rumelhart88].
Support vector machine (SVM)	A discriminative classifier that can also do regression. A distance function between any two data points in a higher-dimensional space is defined. (Projecting data into higher dimensions makes the data more likely to be linearly separable.) The algorithm learns separating hyperplanes that maximally separate the classes in the higher dimension. It tends to be among the best with limited data, losing out to boosting or random trees only when large data sets are available [Vapnik95].

Using Machine Learning in Vision

In general, all the algorithms in Table 13-1 take as input a data vector made up of many features, where the number of features might well number in the thousands. Suppose

your task is to recognize a certain type of object—for example, a person. The first problem that you will encounter is how to collect and label training data that falls into positive (there is a person in the scene) and negative (no person) cases. You will soon realize that people appear at different scales: their image may consist of just a few pixels, or you may be looking at an ear that fills the whole screen. Even worse, people will often be occluded: a man inside a car; a woman's face; one leg showing behind a tree. You need to define what you actually mean by saying a person is in the scene.

Next, you have the problem of collecting data. Do you collect it from a security camera, go to *http://www.flicker.com* and attempt to find "person" labels, or both (and more)? Do you collect movement information? Do you collect other information, such as whether a gate in the scene is open, the time, the season, the temperature? An algorithm that finds people on a beach might fail on a ski slope. You need to capture the variations in the data: different views of people, different lightings, weather conditions, shadows, and so on.

After you have collected lots of data, how will you label it? You must first decide on what you mean by "label". Do you want to know where the person is in the scene? Are actions (running, walking, crawling, following) important? You might end up with a million images or more. How will you label all that? There are many tricks, such as doing background subtraction in a controlled setting and collecting the segmented foreground humans who come into the scene. You can use data services to help in classification; for example, you can pay people to label your images through Amazon's "mechanical turk" (*http://www.mturk.com/mturk/welcome*). If you arrange things to be simple, you can get the cost down to somewhere around a penny per label.

After labeling the data, you must decide which features to extract from the objects. Again, you must know what you are after. If people always appear right side up, there's no reason to use rotation-invariant features and no reason to try to rotate the objects beforehand. In general, you must find features that express some invariance in the objects, such as scale-tolerant histograms of gradients or colors or the popular SIFT features.* If you have background scene information, you might want to first remove it to make other objects stand out. You then perform your image processing, which may consist of normalizing the image (rescaling, rotation, histogram equalization, etc.) and computing many different feature types. The resulting data vectors are each given the label associated with that object, action, or scene.

Once the data is collected and turned into feature vectors, you often want to break up the data into training, validation, and test sets. It is a "best practice" to do your learning, validation, and testing within a cross-validation framework. That is, the data is divided into K subsets and you run many training (possibly validation) and test sessions, where each session consists of different sets of data taking on the roles of training (validation) and test.† The test results from these separate sessions are then averaged to get the final performance result. Cross-validation gives a more accurate picture of how the classifier

* See Lowe's SIFT feature demo (*http://www.cs.ubc.ca/~lowe/keypoints/*).

† One typically does the train (possibly validation) and test cycle five to ten times.

will perform when deployed in operation on novel data. (We'll have more to say about this in what follows.)

Now that the data is prepared, you must choose your classifier. Often the choice of classifier is dictated by computational, data, or memory considerations. For some applications, such as online user preference modeling, you must train the classifier rapidly. In this case, nearest neighbors, normal Bayes, or decision trees would be a good choice. If memory is a consideration, decision trees or neural networks are space efficient. If you have time to train your classifier but it must run quickly, neural networks are a good choice, as are normal Bayes classifiers and support vector machines. If you have time to train but need high accuracy, then boosting and random trees are likely to fit your needs. If you just want an easy, understandable sanity check that your features are chosen well, then decision trees or nearest neighbors are good bets. For best "out of the box" classification performance, try boosting or random trees first.

There is no "best" classifier (see *http://en.wikipedia.org/wiki/No_free_lunch_theorem*). Averaged over all possible types of data distributions, all classifiers perform the same. Thus, we cannot say which algorithm in Table 13-1 is the "best". Over any given data distribution or set of data distributions, however, there is usually a best classifier. Thus, when faced with real data it's a good idea to try many classifiers. Consider your purpose: Is it just to get the right score, or is it to interpret the data? Do you seek fast computation, small memory requirements, or confidence bounds on the decisions? Different classifiers have different properties along these dimensions.

Variable Importance

Two of the algorithms in Table 13-1 allow you to assess a variable's importance.* Given a vector of features, how do you determine the importance of those features for classification accuracy? Binary decision trees do this directly: they are trained by selecting which variable best splits the data at each node. The top node's variable is the most important variable; the next-level variables are the second most important, and so on. Random trees can measure variable importance using a technique developed by Leo Breiman;† this technique can be used with any classifier, but so far it is implemented only for decision and random trees in OpenCV.

One use of variable importance is to reduce the number of features your classifier must consider. Starting with many features, you train the classifier and then find the importance of each feature relative to the other features. You can then discard unimportant features. Eliminating unimportant features improves speed performance (since it eliminates the processing it took to compute those features) and makes training and testing quicker. Also, if you don't have enough data, which is often the case, then eliminating

* This is known as "variable importance" even though it refers to the importance of a variable (noun) and not the fluctuating importance (adjective) of a variable.

† Breiman's variable importance technique is described in "Looking Inside the Black Box" (*www.stat.berkeley.edu/~breiman/wald2002-2.pdf*).

unimportant variables can increase classification accuracy; this yields faster processing with better results.

Breiman's variable importance algorithm runs as follows.

1. Train a classifier on the training set.
2. Use a validation or test set to determine the accuracy of the classifier.
3. For every data point and a chosen feature, randomly choose a new value for that feature from among the values the feature has in the rest of the data set (called "sampling with replacement"). This ensures that the distribution of that feature will remain the same as in the original data set, but now the actual structure or meaning of that feature is erased (because its value is chosen at random from the rest of the data).
4. Train the classifier on the altered set of training data and then measure the accuracy of classification on the altered test or validation data set. If randomizing a feature hurts accuracy a lot, then that feature is very important. If randomizing a feature does not hurt accuracy much, then that feature is of little importance and is a candidate for removal.
5. Restore the original test or validation data set and try the next feature until we are done. The result is an ordering of each feature by its importance.

This procedure is built into random trees and decision trees. Thus, you can use random trees or decision trees to decide which variables you will actually use as features; then you can use the slimmed-down feature vectors to train the same (or another) classifier.

Diagnosing Machine Learning Problems

Getting machine learning to work well can be more of an art than a science. Algorithms often "sort of" work but not quite as well as you need them to. That's where the art comes in; you must figure out what's going wrong in order to fix it. Although we can't go into all the details here, we'll give an overview of some of the more common problems you might encounter.* First, some rules of thumb: More data beats less data, and better features beat better algorithms. If you design your features well—maximizing their independence from one another and minimizing how they vary under different conditions—then almost any algorithm will work well. Beyond that, there are two common problems:

Bias
: Your model assumptions are too strong for the data, so the model won't fit well.

Variance
: Your algorithm has memorized the data *including* the noise, so it can't generalize.

Figure 13-1 shows the basic setup for statistical machine learning. Our job is to model the true function *f* that transforms the underlying inputs to some output. This function may

* Professor Andrew Ng at Stanford University gives the details in a web lecture entitled "Advice for Applying Machine Learning" (*http://www.stanford.edu/class/cs229/materials/ML-advice.pdf*).

be a regression problem (e.g., predicting a person's age from their face) or a category prediction problem (e.g., identifying a person given their facial features). For problems in the real world, noise and unconsidered effects can cause the observed outputs to differ from the theoretical outputs. For example, in face recognition we might learn a model of the measured distance between eyes, mouth, and nose to identify a face. But lighting variations from a nearby flickering bulb might cause noise in the measurements, or a poorly manufactured camera lens might cause a systematic distortion in the measurements that wasn't considered as part of the model. These affects will cause accuracy to suffer.

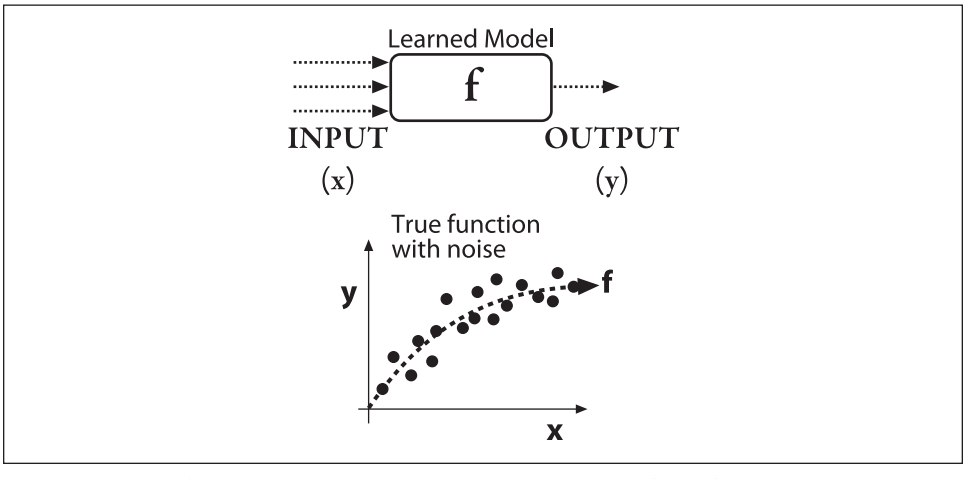

Figure 13-1. Setup for statistical machine learning: we train a classifier to fit a data set; the true model f is almost always corrupted by noise or unknown influences

Figure 13-2 shows under- and overfitting of data in the upper two panels and the consequences in terms of error with training set size in the lower two panels. On the left side of Figure 13-2 we attempt to train a classifier to predict the data in the lower panel of Figure 13-1. If we use a model that's too restrictive—indicated here by the heavy, straight dashed line—then we can never fit the underlying true parabola *f* indicated by the thinner dashed line. Thus, the fit to both the training data and the test data will be poor, even with a lot of data. In this case we have bias because both training and test data are predicted poorly. On the right side of Figure 13-2 we fit the training data exactly, but this produces a nonsense function that fits every bit of noise. Thus, it memorizes the training data as well as the noise in that data. Once again, the resulting fit to the test data is poor. Low training error combined with high test error indicates a variance (overfit) problem.

Sometimes you have to be careful that you are solving the correct problem. If your training and test set error are low but the algorithm does not perform well in the real world, the data set may have been chosen from unrealistic conditions—perhaps because these conditions made collecting or simulating the data easier. If the algorithm just cannot reproduce the test or training set data, then perhaps the algorithm is the wrong one to use or the features that were extracted from the data are ineffective or the "signal" just isn't in the data you collected. Table 13-2 lays out some possible fixes to the problems

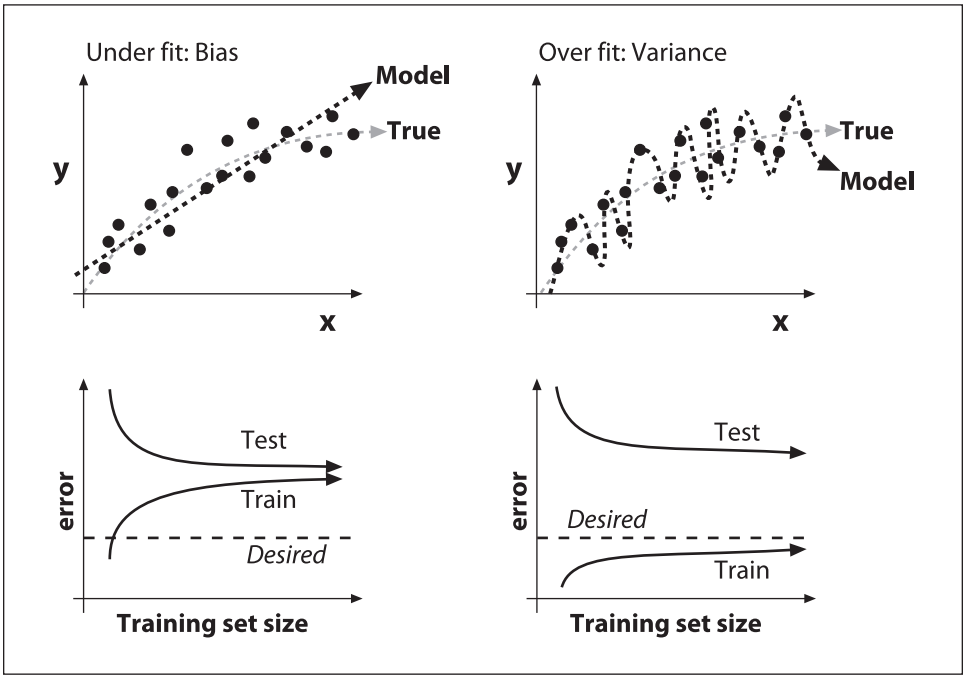

Figure 13-2. Poor model fitting in machine learning and its effect on training and test prediction performance, where the true function is graphed by the lighter dashed line at top: an underfit model for the data (upper left) yields high error in predicting the training and the test set (lower left), whereas an overfit model for the data (upper right) yields low error in the training data but high error in the test data (lower right)

we've described here. Of course, this is not a complete list of the possible problems or solutions. It takes careful thought and design of what data to collect and what features to compute in order for machine learning to work well. It can also take some systematic thinking to diagnose machine learning problems.

Table 13-2. Problems encountered in machine learning and possible solutions to try; coming up with better features will help any problem

Problem	Possible Solutions
Bias	• More features can help make a better fit. • Use a more powerful algorithm.
Variance	• More training data can help smooth the model. • Fewer features can reduce overfitting. • Use a less powerful algorithm.
Good test/train, bad real world	• Collect a more realistic set of data.
Model can't learn test or train	• Redesign features to better capture invariance in the data. • Collect new, more relevant data. • Use a more powerful algorithm.

Cross-validation, bootstrapping, ROC curves, and confusion matrices

Finally, there are some basic tools that are used in machine learning to measure results. In supervised learning, one of the most basic problems is simply knowing how well your algorithm has performed: How accurate is it at classifying or fitting the data? You might think: "Easy, I'll just run it on my test or validation data and get the result." But for real problems, we must account for noise, sampling fluctuations, and sampling errors. Simply put, your test or validation set of data might not accurately reflect the actual distribution of data. To get closer to "guessing" the true performance of the classifier, we employ the technique of *cross-validation* and/or the closely related technique of *bootstrapping*.*

In its most basic form, cross-validation involves dividing the data into K different subsets of data. You train on $K - 1$ of the subsets and test on the final subset of data (the "validation set") that wasn't trained on. You do this K times, where each of the K subsets gets a "turn" at being the validation set, and then average the results.

Bootstrapping is similar to cross-validation, but the validation set is selected at random from the training data. Selected points for that round are used only in test, not training. Then the process starts again from scratch. You do this N times, where each time you randomly select a new set of validation data and average the results in the end. Note that this means some and/or many of the data points are reused in different validation sets, but the results are often superior compared to cross-validation.

Using either one of these techniques can yield more accurate measures of actual performance. This increased accuracy can in turn be used to tune parameters of the learning system as you repeatedly change, train, and measure.

Two other immensely useful ways of assessing, characterizing, and tuning classifiers are plotting the *receiver operating characteristic* (ROC) and filling in a confusion matrix; see Figure 13-3. The ROC curve measures the response over the performance parameter of the classifier over the full range of settings of that parameter. Let's say the parameter is a threshold. Just to make this more concrete, suppose we are trying to recognize yellow flowers in an image and that we have a threshold on the color yellow as our detector. Setting the yellow threshold extremely high would mean that the classifier would fail to recognize any yellow flowers, yielding a false positive rate of 0 but at the cost of a true positive rate also at 0 (lower left part of the curve in Figure 13-3). On the other hand, if the yellow threshold is set to 0 then any signal at all counts as a recognition. This means that all of the true positives (the yellow flowers) are recognized as well as all the false positives (orange and red flowers); thus we have a false positive rate of 100% (upper right part of the curve in Figure 13-3). The best possible ROC curve would be one that follows the y-axis up to 100% and then cuts horizontally over to the upper right corner. Failing that, the closer the curve comes to the upper left corner, the better. One can compute the fraction of area under the ROC curve versus the total area of the ROC plot as a summary statistic of merit: The closer that ratio is to 1 the better is the classifier.

* For more information on these techniques, see "What Are Cross-Validation and Bootstrapping?" (*http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-12.html*).

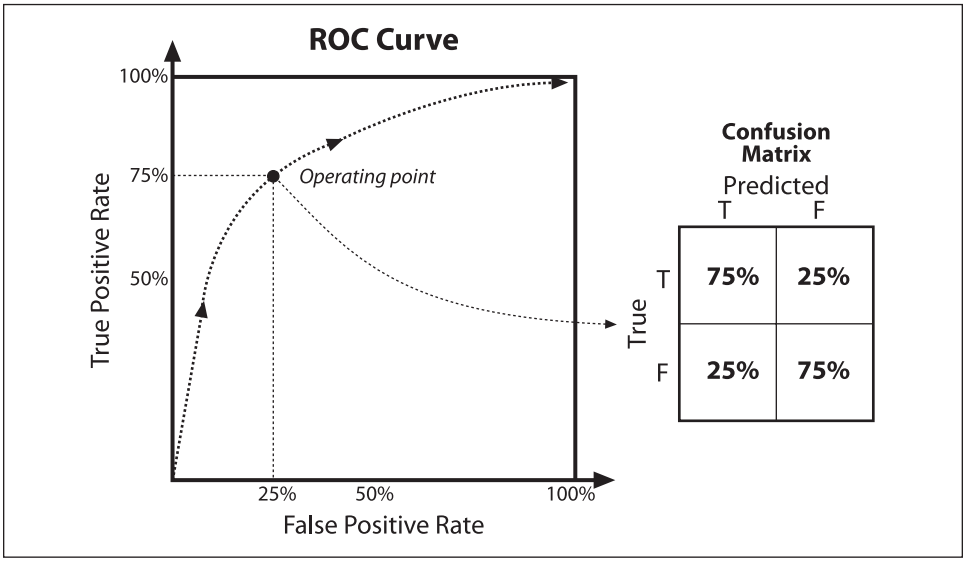

Figure 13-3. Receiver operating curve (ROC) and associated confusion matrix: the former shows the response of correct classifications to false positives along the full range of varying a performance parameter of the classifier; the latter shows the false positives (false recognitions) and false negatives (missed recognitions)

Figure 13-3 also shows a *confusion matrix*. This is just a chart of true and false positives along with true and false negatives. It is another quick way to assess the performance of a classifier: ideally we'd see 100% along the NW-SE diagonal and 0% elsewhere. If we have a classifier that can learn more than one class (e.g., a multilayer perceptron or random forest classifier can learn many different class labels at once), then the confusion matrix generalizes to many classes and you just keep track of the class to which each labeled data point was assigned.

Cost of misclassification. One thing we haven't discussed much here is the cost of misclassification. That is, if our classifier is built to detect poisonous mushrooms (we'll see an example that uses such a data set shortly) then we are willing to have more false negatives (edible mushrooms mistaken as poisonous) as long as we minimize false positives (poisonous mushrooms mistaken as edible). The ROC curve can help with this; we can set our ROC parameter to choose an operation point lower on the curve—toward the lower left of the graph in Figure 13-3. The other way of doing this is to weight false positive errors more than false negatives when generating the ROC curve. For example, you can set each false positive error to count as much as ten false negatives.* Some OpenCV machine learning algorithms, such as decision trees and SVM, can regulate this balance of "hit rate versus false alarm" by specifying prior probabilities of the classes themselves

* This is useful if you have some specific a priori notion of the relative cost of the two error types. For example, the cost of misclassifying one product as another in a supermarket checkout would be easy to quantify exactly beforehand.

(which classes are expected to be more likely and which less) or by specifying weights of the individual training samples.

Mismatched feature variance. Another common problem with training some classifiers arises when the feature vector comprises features of widely different variances. For instance, if one feature is represented by lowercase ASCII characters then it ranges over only 26 different values. In contrast, a feature that is represented by the count of biological cells on a microscope slide might vary over several billion values. An algorithm such as K-nearest neighbors might then see the first feature as relatively constant (nothing to learn from) compared to the cell-count feature. The way to correct this problem is to preprocess each feature variable by normalizing for its variance. This practice is acceptable provided the features are not correlated with each other; when features are correlated, you can normalize by their average variance or by their covariance. Some algorithms, such as decision trees,* are not adversely affected by widely differing variance and so this precaution need not be taken. A rule of thumb is that if the algorithm depends in some way on a distance measure (e.g., weighted values) then you should normalize for variance. One may normalize all features at once and account for their covariance by using the Mahalanobis distance, which is discussed later in this chapter.†

We now turn to discussing some of the machine learning algorithms supported in OpenCV, most of which are found in the *…/opencv/ml* directory. We start with some of the class methods that are universal across the ML sublibrary.

Common Routines in the ML Library

This chapter is written to get you up and running with the machine learning algorithms. As you try out and become comfortable with different methods, you'll also want to reference the *…/opencv/docs/ref/opencvref_ml.htm* manual that installs with OpenCV and/or the online OpenCV Wiki documentation (*http://opencvlibrary.sourceforge.net/*). *Because* this portion of the library is under active development, you will want to know about the latest and greatest available tools.

All the routines in the ML library‡ are written as C++ classes and all derived from the `CvStatModel` class, which holds the methods that are universal to all the algorithms. These methods are listed in Table 13-3. Note that in the `CvStatModel` there are two ways of storing and recalling the model from disk: `save()` versus `write()` and `load()` versus `read()`. For machine learning models, you should use the much simpler `save()`

* Decision trees are not affected by variance differences in feature variables because each variable is searched only for effective separating thresholds. In other words, it doesn't matter how large the variable's range is as long as a clear separating value can be found.

† Readers familiar with machine learning or signal processing might recognize this as a technique for "whitening" the data.

‡ Note that the Haar classifier, Mahalanobis, and K-means algorithms were written before the ML library was created and so are in *cv* and *cvcore* libraries instead.

and load(), which essentially wrap the more complex write() and read() functions into an interface that writes and reads XML and YAML to and from disk. Beyond that, for learning from data the two most important functions, predict() and train(), vary by algorithm and will be discussed next.

Table 13-3. Base class methods for the machine learning (ML) library

CvStatModel:: Methods	Description
`save( const char* filename, const char* name = 0 )`	Saves learned model in XML or YMAL. Use this method for storage.
`load( const char* filename, const char* name=0 );`	Calls `clear()` and then loads XML or YMAL model. Use this method for recall.
`clear()`	De-allocates all memory. Ready for reuse.
`bool train( –data points–, [flags] –responses–, [flags etc] ) ;`	The training function to learn a model of the dataset. Training is specific to the algorithm and so the input parameters will vary.
`float predict( const CvMat* sample [,<prediction_params>] ) const;`	After training, use this function to predict the label or value of a new training point or points.
Constructor, Destructor:	
`CvStatModel(); CvStatModel( const CvMat* train_data ... );`	Default constructor and constructor that allows creation and training of the model in one shot.
`CvStatModel::~CvStatModel();`	The destructor of the ML model.
Write/Read support (but use save/load above instead):	
`write( CvFileStorage* storage, const char* name );`	Generic `CvFileStorage` structured write to disk, located in the *cvcore* library (discussed in Chapter 3) and called by `save()`.
`read( CvFileStorage* storage, CvFileNode* node );`	Generic file read to `CvFileStorage` structure, located in the *cvcore* library and called by `load()`.

Training

The training prototype is as follows:

```
bool CvStatModel::train(
  const CvMat* train_data,
  [int tflag,]                ...,
  const CvMat* responses,     ...,
```

```
    [const CvMat* var_idx,]    ...,
    [const CvMat* sample_idx,] ...,
    [const CvMat* var_type,]   ...,
    [const CvMat* missing_mask,]
    <misc_training_alg_params> ...
);
```

The `train()` method for the machine learning algorithms can assume different forms according to what the algorithm can do. All algorithms take a `CvMat` matrix pointer as training data. This matrix must be of type `32FC1` (32-bit, floating-point, single-channel). `CvMat` does allow for multichannel images, but machine learning algorithms take only a single channel—that is, just a two-dimensional matrix of numbers. Typically this matrix is organized as rows of data points, where each "point" is represented as a vector of features. Hence the columns contain the individual features for each data point and the data points are stacked to yield the 2D single-channel training matrix. To belabor the topic: the typical data matrix is thus composed of (rows, columns) = (data points, features). However, some algorithms can handle transposed matrices directly. For such algorithms you may use the `tflag` parameter to tell the algorithm that the training points are organized in columns. This is just a convenience so that you won't have to transpose a large data matrix. When the algorithm can handle both row-order and column-order data, the following flags apply.

`tflag = CV_ROW_SAMPLE`
: Means that the feature vectors are stored as rows (default)

`tflag = CV_COL_SAMPLE`
: Means that the feature vectors are stored as columns

The reader may well ask: What if my training data is not floating-point numbers but instead is letters of the alphabet or integers representing musical notes or names of plants? The answer is: Fine, just turn them into unique 32-bit floating-point numbers when you fill the `CvMat`. If you have letters as features or labels, you can cast the ASCII character to floats when filling the data array. The same applies to integers. As long as the conversion is unique, things should work—but remember that some routines are sensitive to widely differing variances among features. It's generally best to normalize the variance of features as discussed previously. With the exception of the tree-based algorithms (decision trees, random trees, and boosting) that support both categorical and ordered input variables, all other OpenCV ML algorithms work only with ordered inputs. A popular technique for making ordered-input algorithms also work with categorical data is to represent them in 1-radix notation; for example, if the input variable color may have seven different values then it may be replaced by seven binary variables, where one and only one of the variables may be set to 1.

The parameter responses are either categorical labels such as "poisonous" or "nonpoisonous", as with mushroom identification, or are regression values (numbers) such as body temperatures taken with a thermometer. The response values or "labels" are usually a one-dimensional vector of one value per data point—except for neural networks,

which can have a vector of responses for each data point. Response values are one of two types: For categorical responses, the type can be integer (`32SC1`); for regression values, the response is 32-bit floating-point (`32FC1`). Observe also that some algorithms can deal only with classification problems and others only with regression; but others can handle both. In this last case, the type of output variable is passed either as a separate parameter or as a last element of a `var_type` vector, which can be set as follows.

`CV_VAR_CATEGORICAL`
: Means that the output values are discrete class labels

`CV_VAR_ORDERED` *(=* `CV_VAR_NUMERICAL`*)*
: Means that the output values are ordered; that is, different values can be compared as numbers and so this is a regression problem

The types of input variables can also be specified using `var_type`. However, algorithms of the regression type can handle only ordered-input variables. Sometimes it is possible to make up an ordering for categorical variables as long as the order is kept consistent, but this can sometimes cause difficulties for regression because the pretend "ordered" values may jump around wildly when they have no physical basis for their imposed order.

Many models in the ML library may be trained on a selected feature subset and/or on a selected sample subset of the training set. To make this easier for the user, the method `train()` usually includes the vectors `var_idx` and `sample_idx` as parameters. These may be defaulted to "use all data" by passing `NULL` values for these parameters, but `var_idx` can be used to indentify variables (features) of interest and `sample_idx` can identify data points of interest. Using these, you may specify which features and which sample points on which to train. Both vectors are either single-channel integer (`CV_32SC1`) vectors—that is, lists of zero-based indices—or single-channel 8-bit (`CV_8UC1`) masks of active variables/samples, where a nonzero value signifies active. The parameter `sample_idx` is particularly helpful when you've read in a chunk of data and want to use some of it for training and some of it for test *without* breaking it into two different vectors.

Additionally, some algorithms can handle missing measurements. For example, when the authors were working with manufacturing data, some measurement features would end up missing during the time that workers took coffee breaks. Sometimes experimental data simply is forgotten, such as forgetting to take a patient's temperature one day during a medical experiment. For such situations, the parameter `missing_mask`, an 8-bit matrix of the same dimensions as `train_data`, is used to mark the missed values (nonzero elements of the mask). Some algorithms cannot handle missing values, so the missing points should be interpolated by the user before training or the corrupted records should be rejected in advance. Other algorithms, such as decision tree and naïve Bayes, handle missing values in different ways. Decision trees use alternative splits (called "surrogate splits" by Breiman); the naïve Bayes algorithm infers the values.

Usually, the previous model state is cleared by `clear()` before running the training procedure. However, some algorithms may optionally update the model learning with the new training data instead of starting from scratch.

Prediction

When using the method predict(), the var_idx parameter that specifies which features were used in the train() method is remembered and then used to extract only the necessary components from the input sample. The general form of the predict() method is as follows:

```
float CvStatMode::predict(
  const CvMat* sample
  [, <prediction_params>]
) const;
```

This method is used to predict the response for a new input data vector. When using a classifier, predict() returns a class label. For the case of regression, this method returns a numerical value. Note that the input sample must have as many components as the train_data that was used for training. Additional prediction_params are algorithm-specific and allow for such things as missing feature values in tree-based methods. The function suffix const tells us that prediction does not affect the internal state of the model, so this method is thread-safe and can be run in parallel, which is useful for web servers performing image retrieval for multiple clients and for robots that need to accelerate the scanning of a scene.

Controlling Training Iterations

Although the iteration control structure CvTermCriteria has been discussed in other chapters, it is used by several machine learning routines. So, just to remind you of what the function is, we repeat it here.

```
typedef struct CvTermCriteria {
    int    type;     /* CV_TERMCRIT_ITER and/or CV_TERMCRIT_EPS */
    int    max_iter; /* maximum number of iterations */
    double epsilon;  /* stop when error is below this value  */
}
```

The integer parameter max_iter sets the total number of iterations that the algorithm will perform. The epsilon parameter sets an error threshold stopping criteria; when the error drops below this level, the routine stops. Finally, the type tells which of these two criteria to use, though you may add the criteria together and so use both (CV_TERMCRIT_ITER | CV_TERMCRIT_EPS). The defined values for term_crit.type are:

```
#define CV_TERMCRIT_ITER    1
#define CV_TERMCRIT_NUMBER  CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS     2
```

Let's now move on to describing specific algorithms that are implemented in OpenCV. We will start with the frequently used Mahalanobis distance metric and then go into some detail on one unsupervised algorithm (K-means); both of these may be found in the *cxcore* library. We then move into the machine learning library proper with the normal Bayes classifier, after which we discuss decision-tree algorithms (decision trees, boosting, random trees, and Haar cascade). For the other algorithms we'll provide short descriptions and usage examples.

Mahalanobis Distance

The *Mahalanobis distance* is a distance measure that accounts for the covariance or "stretch" of the space in which the data lies. If you know what a *Z-score* is then you can think of the Mahalanobis distance as a multidimensional analogue of the *Z*-score. Figure 13-4(a) shows an initial distribution between three sets of data that make the vertical sets look closer together. When we normalize the space by the covariance in the data, we see in Figure 13-4(b) that that horizontal data sets are actually closer together. This sort of thing occurs frequently; for instance, if we are comparing people's height in meters with their age in days, we'd see very little variance in height to relate to the large variance in age. By normalizing for the variance we can obtain a more realistic comparison of variables. Some classifiers such as K-nearest neighbors deal poorly with large differences in variance, whereas other algorithms (such as decision trees) don't mind it.

We can already get a hint for what the Mahalanobis distance must be by looking at Figure 13-4;* we must somehow divide out the covariance of the data while measuring distance. First, let us review what covariance is. Given a list X of N data points, where each data point may be of dimension (vector length) K with mean vector μ_x (consisting of individual means $\mu_{1,\ldots,K}$), the covariance is a K-by-K matrix given by:

$$\sum = E[(X - \mu_x)(X - \mu_x)^{\mathrm{T}}]$$

where $E[\cdot]$ is the expectation operator. OpenCV makes computing the covariance matrix easy, using

```
void cvCalcCovarMatrix(
  const CvArr** vects,
  int           count,
  CvArr*        cov_mat,
  CvArr*        avg,
  int           flags
);
```

This function is a little bit tricky. Note that `vects` is a pointer to a pointer of `CvArr`. This implies that we have `vects[0]` through `vects[count-1]`, but it actually depends on the `flags` settings as described in what follows. Basically, there are two cases.

1. `Vects` is a 1D vector of pointers to 1D vectors or 2D matrices (the two dimensions are to accommodate images). That is, each `vects[i]` can point to a 1D or a 2D vector, which occurs if neither `CV_COV_ROWS` nor `CV_COV_COLS` is set. The accumulating covariance computation is scaled or divided by the number of data points given by `count` if `CV_COVAR_SCALE` is set.
2. Often there is only one input vector, so use only `vects[0]` if either `CV_COVAR_ROWS` or `CV_COVAR_COLS` is set. If this is set, then scaling by the value given by `count` is ignored

* Note that Figure 13-4 has a diagonal covariance matrix, which entails independent X and Y variance rather than actual covariance. This was done to make the explanation simple. In reality, data is often "stretched" in much more interesting ways.

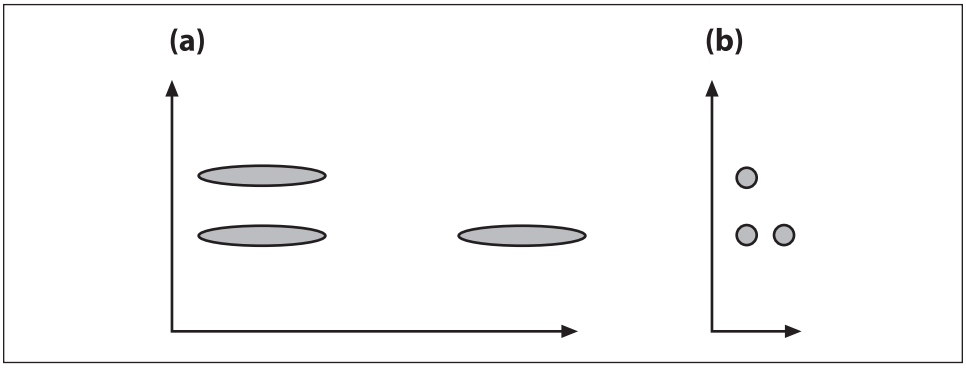

Figure 13-4. The Mahalanobis computation allows us to reinterpret the data's covariance as a "stretch" of the space: (a) the vertical distance between raw data sets is less than the horizontal distance; (b) after the space is normalized for variance, the horizontal distance between data sets is less than the vertical distance

in favor of the number of actual data vectors contained in `vects[0]`. All the data points are then in:

a. the rows of `vects[0]` if `CV_COVAR_ROWS` is set; or

b. the columns of `vects[0]` if instead `CV_COVAR_COLS` is set. You cannot set both row and column flags simultaneously (see flag descriptions for more details).

`Vects` can be of types `8UC1`, `16UC1`, `32FC1`, or `64FC1`. In any case, `vects` contains a list of *K*-dimensional data points. To reiterate: `count` is how many vectors there are in `vects[]` for case 1 (`CV_COVAR_ROWS` and `CV_COVAR_COLS` not set); for case 2a and 2b (`CV_COVAR_ROWS` or `CV_COVAR_COLS` is set), `count` is ignored and the actual number of vectors in `vects[0]` is used instead. The resulting *K*-by-*K* covariance matrix will be returned in `cov_mat`, and it can be of type `CV_32FC1` or `CV_64FC1`. Whether or not the vector `avg` is used depends on the settings of `flags` (see listing that follows). If `avg` is used then it has the same type as `vects` and contains the *K*-feature averages across `vects`. The parameter `flags` can have many combinations of settings formed by adding values together (for more complicated applications, refer to the *…/opencv/docs/ref/opencvref_cxcore.htm* documentation). In general, you will set `flags` to one of the following.

`CV_COVAR_NORMAL`
: Do the regular type of covariance calculation as in the previously displayed equation. Average the results by the number in `count` if `CV_COVAR_SCALE` is not set; otherwise, average by the number of data points in `vects[0]`.

`CV_COVAR_SCALE`
: Normalize the computed covariance matrix.

`CV_COVAR_USE_AVG`
: Use the `avg` matrix instead of automatically calculating the average of each feature. Setting this saves on computation time if you already have the averages (e.g., by

having called cvAvg() yourself); otherwise, the routine will compute these averages for you.*

Most often you will combine your data into one big matrix, let's say by rows of data points; then flags would be set as `flags = CV_COVAR_NORMAL | CV_COVAR_SCALE | CV_COVAR_ROWS`.

We now have the covariance matrix. For Mahalanobis distance, however, we'll need to divide out the variance of the space and so will need the *inverse* covariance matrix. This is easily done by using:

```
double cvInvert(
  const CvArr* src,
  CvArr*       dst,
  int          method = CV_LU
);
```

In cvInvert(), the src matrix should be the covariance matrix calculated before and dst should be a same sized matrix, which will be filled with the inverse on return. You could leave the method at its default value, CV_LU, but it is better to set the method to CV_SVD_SYM.†

With the inverse covariance matrix Σ^{-1} finally in hand, we can move on to the Mahalanobis distance measure. This measure is much like the Euclidean distance measure, which is the square root of the sum of squared differences between two vectors **x** and **y**, but it divides out the covariance of the space:

$$d_{\text{Mahalanobis}}(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}\,\Sigma^{-1}\,(\mathbf{x}-\mathbf{y})}$$

This distance is just a number. Note that if the covariance matrix is the identity matrix then the Mahalanobis distance is equal to the Euclidean distance. We finally arrive at the actual function that computes the Mahalanobis distance. It takes two input vectors (vec1 and vec2) and the inverse covariance in mat, and it returns the distance as a double:

```
double cvMahalanobis(
  const CvArr* vec1,
  const CvArr* vec2,
  CvArr*       mat
);
```

The Mahalanobis distance is an important measure of similarity between two different data points in a multidimensional space, but is not a clustering algorithm or classifier itself. Let us now move on, starting with the most frequently used clustering algorithm: K-means.

* A precomputed average data vector should be passed if the user has a more statistically justified value of the average or if the covariance matrix is computed by blocks.

† CV_SVD could also be used in this case, but it is somewhat slower and less accurate than CV_SVD_SYM. CV_SVD_SYM, even if it is slower than CV_LU, still should be used if the dimensionality of the space is much smaller than the number of data points. In such a case the overall computing time will be dominated by cvCalcCovarMatrix() anyway. So it may be wise to spend a little bit more time on computing inverse covariance matrix more accurately (much more accurately, if the set of points is concentrated in a subspace of a smaller dimensionality). Thus, CV_SVD_SYM is usually the best choice for this task.

K-Means

K-means is a clustering algorithm implemented in the *cxcore* because it was written long before the ML library. K-means attempts to find the natural clusters or "clumps" in the data. The user sets the desired number of clusters and then K-means rapidly finds a good placement for those cluster centers, where "good" means that the cluster centers tend to end up located in the middle of the natural clumps of data. It is one of the most used clustering techniques and has strong similarities to the expectation maximization algorithm for Gaussian mixture (implemented as `CvEM()` in the ML library) as well as some similarities to the mean-shift algorithm discussed in Chapter 9 (implemented as `cvMeanShift()` in the CV library). K-means is an iterative algorithm and, as implemented in OpenCV, is also known as Lloyd's algorithm* or (equivalently) "Voronoi iteration". The algorithm runs as follows.

1. Take as input (a) a data set and (b) desired number of clusters K (chosen by the user).
2. Randomly assign cluster center locations.
3. Associate each data point with its nearest cluster center.
4. Move cluster centers to the centroid of their data points.
5. Return to step 3 until convergence (centroid does not move).

Figure 13-5 diagrams K-means in action; in this case, it takes just two iterations to converge. In real cases the algorithm often converges rapidly, but it can sometimes require a large number of iterations.

Problems and Solutions

K-means is an extremely effective clustering algorithm, but it does have three problems.

1. K-means isn't guaranteed to find the best possible solution to locating the cluster centers. However, it is guaranteed to converge to some solution (i.e., the iterations won't continue indefinitely).
2. K-means doesn't tell you how many cluster centers you should use. If we had chosen two or four clusters for the example of Figure 13-5, then the results would be different and perhaps nonintuitive.
3. K-means presumes that the covariance in the space either doesn't matter or has already been normalized (cf. our discussion of the Mahalanobis distance).

Each one of these problems has a "solution", or at least an approach that helps. The first two of these solutions depend on "explaining the variance of the data". In K-means, each cluster center "owns" its data points and we compute the variance of those points.

* S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory 28* (1982), 129–137.

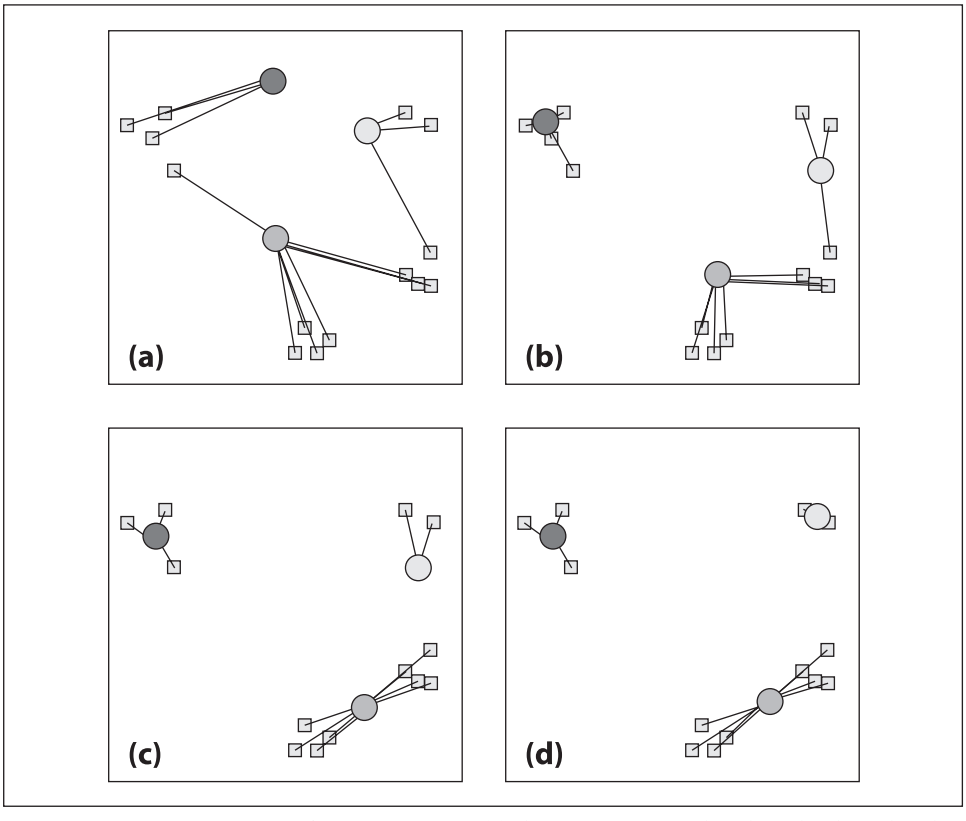

Figure 13-5. K-means in action for two iterations: (a) cluster centers are placed randomly and each data point is then assigned to its nearest cluster center; (b) cluster centers are moved to the centroid of their points; (c) data points are again assigned to their nearest cluster centers; (d) cluster centers are again moved to the centroid of their points

The best clustering minimizes the variance without causing too much complexity (too many clusters). With that in mind, the listed problems can be ameliorated as follows.

1. Run K-means several times, each with different placement of the cluster centers (easy to do, since OpenCV places the centers at random); then choose the run whose results exhibit the least variance.
2. Start with one cluster and try an increasing number of clusters (up to some limit), each time employing the method of #1 as well. Usually the total variance will shrink quite rapidly, after which an "elbow" will appear in the variance curve; this indicates that a new cluster center does not significantly reduce the total variance. Stop at the elbow and keep that many cluster centers.
3. Multiply the data by the inverse covariance matrix (as described in the "Mahalanobis Distance" section). For example, if the input data vectors D are organized as rows with one data point per row, then normalize the "stretch" in the space by computing a new data vector D^*, where $D^* = D\Sigma^{-1/2}$.

K-Means Code

The call for K-means is simple:

```
void cvKMeans2(
  const CvArr*   samples,
  int            cluster_count,
  CvArr*         labels,
  CvTermCriteria termcrit
);
```

The `samples` array is a matrix of multidimensional data points, one per row. There is a little subtlety here in that each element of the data point may be either a regular floating-point vector of `CV_32FC1` numbers or a multidimensional point of type `CV_32FC2` or `CV_32FC3` or even `CV_32FC(K)`.* The parameter `cluster_count` is simply how many clusters you want, and the return vector `labels` contains the final cluster index for each data point. We encountered `termcrit` in the section "Common Routines in the ML Library" and in the "Controlling Training Iterations" subsection.

It's instructive to see a complete example of K-means in code (Example 13-1), because the data generation sections can be used to test other machine learning routines.

Example 13-1. Using K-means

```
#include "cxcore.h"
#include "highgui.h"

void main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    CvScalar color_tab[MAX_CLUSTERS];
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG(0xffffffff);

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );

    for(;;)
    {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        /* generate random sample from multivariate
```

* This is exactly equivalent to an *N*-by-*K* matrix in which the *N* rows are the data points, the *K* columns are the individual components of each point's location, and the underlying data type is `32FC1`. Recall that, owing to the memory layout used for arrays, there is no distinction between these representations.

Example 13-1. Using K-means (continued)

```
            Gaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;
            cvGetRows( points, &point_chunk,
                       k*sample_count/cluster_count,
                       k == cluster_count - 1 ? sample_count :
                       (k+1)*sample_count/cluster_count );
            cvRandArr( &rng, &point_chunk, CV_RAND_NORMAL,
                       cvScalar(center.x,center.y,0,0),
                       cvScalar(img->width/6, img->height/6,0,0) );
        }

        /* shuffle samples */
        for( i = 0; i < sample_count/2; i++ )
        {
            CvPoint2D32f* pt1 = (CvPoint2D32f*)points->data.fl +
                                cvRandInt(&rng)%sample_count;
            CvPoint2D32f* pt2 = (CvPoint2D32f*)points->data.fl +
                                cvRandInt(&rng)%sample_count;
            CvPoint2D32f temp;
            CV_SWAP( *pt1, *pt2, temp );
        }

        cvKMeans2( points, cluster_count, clusters,
                   cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER,
                                   10, 1.0 ));
        cvZero( img );
        for( i = 0; i < sample_count; i++ )
        {
            CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
            int cluster_idx = clusters->data.i[i];
            cvCircle( img, cvPointFrom32f(pt), 2,
                      color_tab[cluster_idx], CV_FILLED );
        }

        cvReleaseMat( &points );
        cvReleaseMat( &clusters );

        cvShowImage( "clusters", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;
    }
}
```

In this code we included *highgui.h* to use a window output interface and *cxcore.h* because it contains `Kmeans2()`. In `main()`, we set up the coloring of returned clusters for display, set the upper limit to how many cluster centers can be chosen at random to `MAX_`

CLUSTERS (here 5) in cluster_count, and allow up to 1,000 data points, where the random value for this is kept in sample_count. In the outer for{} loop, which repeats until the Esc key is hit, we allocate a floating point matrix points to contain sample_count data points (in this case, a single column of 2D data points CV_32FC2) and allocate an integer matrix clusters to contain their resulting cluster labels, 0 through cluster_count - 1.

We next enter a data generation for{} loop that can be reused for testing other algorithms. For each cluster, we fill in the points array in successive chunks of size sample_count/cluster_count. Each chunk is filled with a normal distribution, CV_RAND_NORMAL, of 2D (CV_32FC2) data points centered on a randomly chosen 2D center.

The next for{} loop merely shuffles the resulting total "pack" of points. We then call cvKMeans2(), which runs until the largest movement of a cluster center is less than 1 (but allowing no more than ten iterations).

The final for{} loop just draws the results. This is followed by de-allocating the allocated arrays and displaying the results in the "clusters" image. Finally, we wait indefinitely (cvWaitKey(0)) to allow the user another run or to quit via the Esc key.

Naïve/Normal Bayes Classifier

The preceding routines are from *cxcore*. We'll now start discussing the machine learning (ML) library section of OpenCV. We'll begin with OpenCV's simplest supervised classifier, CvNormalBayesClassifier, which is called both a *normal Bayes* classifier and a *naïve Bayes* classifier. It's "naïve" because it assumes that all the features are independent from one another even though this is seldom the case (e.g., finding one eye usually implies that another eye is lurking nearby). Zhang discusses possible reasons for the sometimes surprisingly good performance of this classifier [Zhang04]. Naïve Bayes is not used for regression, but it's an effective classifier that can handle multiple classes, not just two. This classifier is the simplest possible case of what is now a large and growing field known as Bayesian networks, or "probabilistic graphical models". Bayesian networks are causal models; in Figure 13-6, for example, the face features in an image are caused by the existence of a face. In use, the face variable is considered a *hidden variable* and the face features—via image processing operations on the input image—constitute the observed evidence for the existence of a face. We call this a *generative* model because the face causally generates the face features. Conversely, we might start by assuming the face node is active and then randomly sample what features are probabilistically generated given that face is active.* This top-down generation of data with the same statistics as the learned causal model (here, the face) is a useful ability that a purely discriminative model does not possess. For example, one might generate faces for computer graphics display, or a robot might literally "imagine" what it should do next by generating scenes, objects, and interactions. In contrast to Figure 13-6, a discriminative model would have the direction of the arrows reversed.

* Generating a face would be silly with the naïve Bayes algorithm because it assumes independence of features. But a more general Bayesian network can easily build in feature dependence as needed.

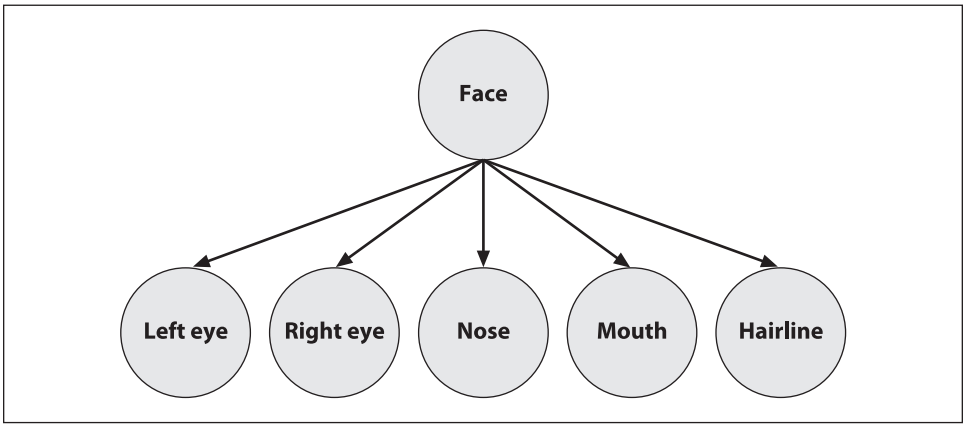

Figure 13-6. A (naïve) Bayesian network, where the lower-level features are caused by the presence of an object (the face)

Bayesian networks are a deep and initially difficult field to understand, but the naïve Bayes algorithm derives from a simple application of Bayes' law. In this case, the probability (denoted *p*) of a face given the features (denoted, left to right in Figure 13-6, as LE, RE, N, M, H) is:

$$p(\text{face}\,|\,\text{LE},\text{RE},\text{N},\text{M},\text{H}) = \frac{p(\text{LE},\text{RE},\text{N},\text{M},\text{H}\,|\,\text{face})\,p(\text{face})}{p(\text{LE},\text{RE},\text{N},\text{M},\text{H})}$$

Just so you'll know, in English this equation means:

$$\text{posterior probability} = \frac{\text{likelihood} \times \text{prior probability}}{\text{evidence}}$$

In practice, we compute some evidence and then decide what object caused it. Since the computed evidence stays the same for the objects, we can drop that term. If we have many models then we need only find the one with the maximum numerator. The numerator is exactly the joint probability of the model with the data: *p*(face, LE, RE, N, M, H). We can then use the definition of conditional probability to derive the joint probability:

$$\begin{aligned} & p(\text{face},\text{LE},\text{RE},\text{N},\text{M},\text{H}) \\ & \quad = p(\text{face})\,p(\text{LE}\,|\,\text{face})\,p(\text{RE}\,|\,\text{face},\text{LE})\,p(\text{N}\,|\,\text{face},\text{LE},\text{RE}) \\ & \qquad \times p(\text{M}\,|\,\text{face},\text{LE},\text{RE},\text{N})\,p(\text{H}\,|\,\text{face},\text{LE},\text{RE},\text{N},\text{M}) \end{aligned}$$

Applying our assumption of independence of features, the conditional features drop out. So, generalizing face to "object" and particular features to "all features", we obtain the reduced equation:

$$p(\text{object, all features}) = p(\text{object}) \prod_{i=1}^{\text{all features}} p(\text{feature}_i\,|\,\text{object})$$

To use this as an overall classifier, we learn models for the objects that we want. In run mode we compute the features and find the object that maximizes this equation. We typically then test to see if the probability for that "winning" object is over a given threshold. If it is, then we declare the object to be found; if not, we declare that no object was recognized.

If (as frequently occurs) there is only one object of interest, then you might ask: "The probability I'm computing is the probability relative to what?" In such cases, there is always an implicit second object—namely, the background—which is everything that is *not* the object of interest that we're trying to learn and recognize.

Learning the models is easy. We take many images of the objects; we then compute features over those objects and compute the fraction of how many times a feature occurred over the training set for each object. In practice, we don't allow zero probabilities because that would eliminate the chance of an object existing; hence zero probabilities are typically set to some very low number. In general, if you don't have much data then simple models such as naïve Bayes will tend to outperform more complex models, which will "assume" too much about the data (bias).

Naïve/Normal Bayes Code

The training method for the normal Bayes classifier is:

```
bool CvNormalBayesClassifier::train(
  const CvMat* _train_data,
  const CvMat* _responses,
  const CvMat* _var_idx    = 0,
  const CvMat* _sample_idx = 0,
  bool         update      = false
);
```

This follows the generic method for training described previously, but it allows only data for which each row is a training point (i.e., as if `tflag=CV_ROW_SAMPLE`). Also, the input `_train_data` is a single-column `CV_32FC1` vector that can only be of type ordered, `CV_VAR_ORDERED` (numbers). The output label `_responses` is a vector column that can only be of categorical type `CV_VAR_CATEGORICAL` (integers, even if contained in a float vector). The parameters `_var_idx` and `_sample_idx` are optional; they allow you to mark (respectively) features and data points that you want to use. Mostly you'll use all features and data and simply pass `NULL` for these vectors, but `_sample_idx` can be used to divide the training and test sets, for example. Both vectors are either single-channel integer (`CV_32SC1`) zero-based indexes or 8-bit (`CV_8UC1`) mask values, where 0 means to skip. Finally, `update` can be set to merely update the normal Bayes learning rather than to learn a new model from scratch.

The prediction for method for `CvNormalBayesClassifier` computes the most probable class for its input vectors. One or more input data vectors are stored as rows of the `samples` matrix. The predictions are returned in corresponding rows of the `results` vector. If there is only a single input in samples, then the resulting prediction is returned

as a float value by the `predict` method and the `results` array may be set to NULL (the default). The format for the prediction method is:

```
float CvNormalBayesClassifier::predict(
  const CvMat* samples,
  CvMat*       results = 0
) const;
```

We move next to a discussion of tree-based classifiers.

Binary Decision Trees

We will go through decision trees in detail, since they are highly useful and use most of the functionality in the machine learning library (and thus serve well as an instructional example). Binary decision trees were invented by Leo Breiman and colleagues,* who named them *classification and regression tree* (CART) algorithms. This is the decision tree algorithm that OpenCV implements. The gist of the algorithm is to define an impurity metric relative to the data in every node of the tree. For example, when using regression to fit a function, we might use the sum of squared differences between the true value and the predicted value. We want to minimize the sum of differences (the "impurity") in each node of the tree. For categorical labels, we define a measure that is minimal when most values in a node are of the same class. Three common measures to use are *entropy*, *Gini index*, and *misclassification* (all are described in this section). Once we have such a metric, a binary decision tree searches through the feature vector to find which feature combined with which threshold most purifies the data. By convention, we say that features above the threshold are "true" and that the data thus classified will branch to the left; the other data points branch right. This procedure is then used recursively down each branch of the tree until the data is of sufficient purity or until the number of data points in a node reaches a set minimum.

The equations for node impurity $i(N)$ are given next. We must deal with two cases, regression and classification.

Regression Impurity

For regression or function fitting, the equation for node impurity is simply the square of the difference in value between the node value y and the data value x. We want to minimize:

$$i(N)=\sum_j (y_j - x_j)^2$$

Classification Impurity

For classification, decision trees often use one of three methods: *entropy impurity*, *Gini impurity*, or *misclassification impurity*. For these methods, we use the notation $P(\omega_j)$ to

* L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees* (1984), Wadsworth.

denote the fraction of patterns at node N that are in class ω_j. Each of these impurities has slightly different effects on the splitting decision. Gini is the most commonly used, but all the algorithms attempt to minimize the impurity at a node. Figure 13-7 graphs the impurity measures that we want to minimize.

Entropy impurity

$$i(N) = -\sum_j P(\omega_j) \log P(\omega_j)$$

Gini impurity

$$i(N) = \sum_{j \neq i} P(\omega_i) P(\omega_j)$$

Misclassification impurity

$$i(N) = 1 - \max P(\omega_j)$$

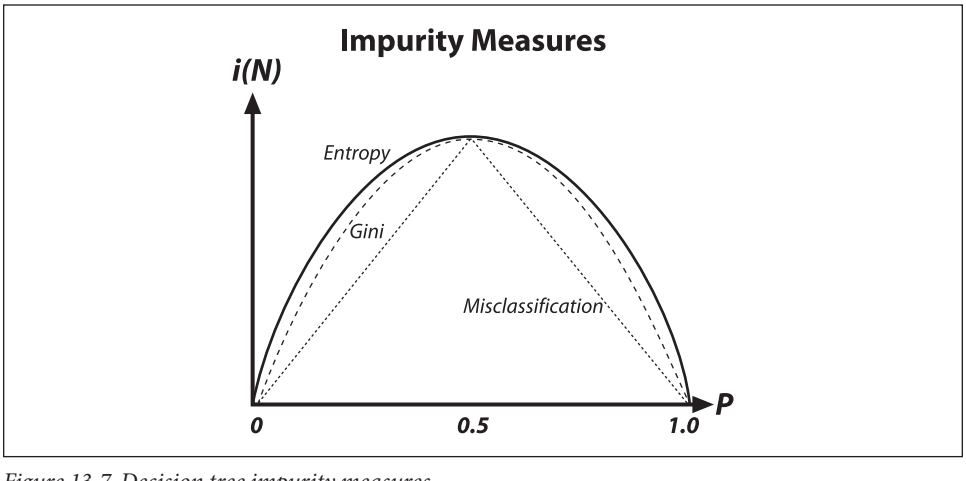

Figure 13-7. Decision tree impurity measures

Decision trees are perhaps the most widely used classification technology. This is due to their simplicity of implementation, ease of interpretation of results, flexibility with different data types (categorical, numerical, unnormalized and mixes thereof), ability to handle missing data through surrogate splits, and natural way of assigning importance to the data features by order of splitting. Decision trees form the basis of other algorithms such as boosting and random trees, which we will discuss shortly.

Decision Tree Usage

In what follows we describe perhaps more than enough for you to get decision trees working well. However, there are many more methods for accessing nodes, modifying splits, and so forth. For that level of detail (which few readers are likely ever to need)

you should consult the user manual *…/opencv/docs/ref/opencvref_ml.htm*, particularly with regard to the classes `CvDTree{}`, the training class `CvDTreeTrainData{}`, and the nodes `CvDTreeNode{}` and splits `CvDTreeSplit{}`.

For a pragmatic introduction, we start by dissecting a specific example. In the *…/opencv/samples/c* directory, there is a *mushroom.cpp* file that runs decision trees on the *agaricus-lepiota.data* data file. This data file consists of a label "p" or "e" (denoting poisonous or edible, respectively) followed by 22 categorical attributes, each represented by a single letter. Observe that the data file is given in "comma separated value" (CSV) format, where the features' values are separated from each other by commas. In *mushroom.cpp* there is a rather messy function `mushroom_read_database()` for reading in this particular data file. This function is rather overspecific and brittle but mainly it's just filling three arrays as follows. (1) A floating-point matrix `data[][]`, which has dimensions rows = number of data points by columns = number of features (22 in this case) and where all the features are converted from their categorical letter values to floating-point numbers. (2) A character matrix `missing[][]`, where a "true" or "1" indicates a missing value that is indicated in the raw data file by a question mark and where all other values are set to 0. (3) A floating-point vector `responses[]`, which contains the poison "p" or edible "e" response cast in floating-point values. In most cases you would write a more general data input program. We'll now discuss the main working points of *mushroom.cpp*, all of which are called directly or indirectly from `main()` in the program.

Training the tree

For training the tree, we fill out the tree parameter structure `CvDTreeParams{}`:

```
struct CvDTreeParams {

  int   max_categories;       //Until pre-clustering
  int   max_depth;            //Maximum levels in a tree
  int   min_sample_count;     //Don't split a node if less
  int   cv_folds;             //Prune tree with K fold cross-validation
  bool  use_surrogates;       //Alternate splits for missing data
  bool  use_1se_rule;         //Harsher pruning
  bool  truncate_pruned_tree; //Don't "remember" pruned branches
  float regression_accuracy;  //One of the "stop splitting" criteria
  const float* priors;        //Weight of each prediction category

  CvDTreeParams() : max_categories(10), max_depth(INT_MAX),
    min_sample_count(10), cv_folds(10), use_surrogates(true),
    use_1se_rule(true), truncate_pruned_tree(true),
    regression_accuracy(0.01f), priors(NULL) { ; }

  CvDTreeParams(
    int          _max_depth,
    int          _min_sample_count,
    float        _regression_accuracy,
    bool         _use_surrogates,
    int          _max_categories,
    int          _cv_folds,
    bool         _use_1se_rule,
```

```
    bool         _truncate_pruned_tree,
    const float* _priors
  );
}
```

In the structure, `max_categories` has a default value of 10. This limits the number of categorical values before which the decision tree will precluster those categories so that it will have to test no more than $2^{\texttt{max_categories}}-2$ possible value subsets.* This isn't a problem for ordered or numerical features, where the algorithm just has to find a threshold at which to split left or right. Those variables that have more categories than `max_categories` will have their category values clustered down to `max_categories` possible values. In this way, decision trees will have to test no more than `max_categories` levels at a time. This parameter, when set to a low value, reduces computation at the cost of accuracy.

The other parameters are fairly self-explanatory. The last parameter, `priors`, can be crucial. It sets the relative weight that you give to misclassification. That is, if the weight of the first category is 1 and the weight of the second category is 10, then each mistake in predicting the second category is equivalent to making 10 mistakes in predicting the first category. In the code we have edible and poisonous mushrooms, so we "punish" mistaking a poisonous mushroom for an edible one 10 times more than mistaking an edible mushroom for a poisonous one.

The template of the methods for training a decision tree is shown below. There are two methods: the first is used for working directly with decision trees; the second is for ensembles (as used in boosting) or forests (as used in random trees).

```
// Work directly with decision trees:
  bool CvDTree::train(
  const CvMat*  _train_data,
  int           _tflag,
  const CvMat*  _responses,
  const CvMat*  _var_idx      = 0,
  const CvMat*  _sample_idx   = 0,
  const CvMat*  _var_type     = 0,
  const CvMat*  _missing_mask = 0,
  CvDTreeParams params        = CvDTreeParams()
);

// Method that ensembles of decision trees use to call individual
```

* More detail on categorical vs. ordered splits: Whereas a split on an ordered variable has the form "if $x < a$ then go left, else go right", a split on a categorical variable has the form "if $x \in \{v_1, v_2, v_3, \ldots, v_k\}$ then go left, else go right", where the v_i are some possible values of the variable. Thus, if a categorical variable has N possible values then, in order to find a best split on that variable, one needs to try $2^N - 2$ subsets (empty and full subset are excluded). Thus, an approximate algorithm is used whereby all N values are grouped into $K \leq$ `max_categories` clusters (via the K-mean algorithm) based on the statistics of the samples in the currently analyzed node. Thereafter, the algorithm tries different combinations of the clusters and chooses the best split, which often gives quite a good result. Note that for the two most common tasks, two-class classification and regression, the optimal categorical split (i.e., the best subset of values) can be found efficiently without any clustering. Hence the clustering is applied only in $n > 2$-class classification problems for categorical variables with $N >$ `max_categories` possible values. Therefore, you should think twice before setting `max_categories` to anything greater than 20, which would imply more than a million operations for each split!

```
// training for each tree in the ensemble
bool CvDTree::train(
  CvDTreeTrainData* _train_data,
  const CvMat*      _subsample_idx
);
```

In the train() method, we have the floating-point _train_data[][] matrix. In that matrix, if _tflag is set to CV_ROW_SAMPLE then each row is a data point consisting of a vector of features that make up the columns of the matrix. If tflag is set to CV_COL_SAMPLE, the row and column meanings are reversed. The _responses[] argument is a floating-point vector of values to be predicted given the data features. The other parameters are optional. The vector _var_idx indicates features to include, and the vector _sample_idx indicates data points to include; both of these vectors are either zero-based integer lists of values to skip or 8-bit masks of active (1) or skip (0) values (see our general discussion of the train() method earlier in the chapter). The byte (CV_8UC1) vector _var_type is a zero-based mask for each feature type (CV_VAR_CATEGORICAL or CV_VAR_ORDERED*); its size is equal to the number of features plus 1. That last entry is for the response type to be learned. The byte-valued _missing_mask[][] matrix is used to indicate missing values with a 1 (else 0 is used). Example 13-2 details the creation and training of a decision tree.

Example 13-2. Creating and training a decision tree

```
float priors[] = { 1.0, 10.0}; // Edible vs poisonous weights

CvMat* var_type;

var_type = cvCreateMat( data->cols + 1, 1, CV_8U );

cvSet( var_type, cvScalarAll(CV_VAR_CATEGORICAL) ); // all these vars
                                                    // are categorical
CvDTree* dtree;
dtree = new CvDTree;
dtree->train(
  data,
  CV_ROW_SAMPLE,
  responses,
  0,
  0,
  var_type,
  missing,
  CvDTreeParams(
    8,      // max depth
    10,     // min sample count
    0,      // regression accuracy: N/A here
    true,   // compute surrogate split,
            //   since we have missing data
    15,     // max number of categories
            //   (use suboptimal algorithm for
            //   larger numbers)
    10,     // cross-validations
```

* CV_VAR_ORDERED is the same thing as CV_VAR_NUMERICAL.

Example 13-2. Creating and training a decision tree (continued)

```
    true,   // use 1SE rule => smaller tree
    true,   // throw away the pruned tree branches
    priors  // the array of priors, the bigger
            //   p_weight, the more attention
            //   to the poisonous mushrooms
  )
);
```

In this code the decision tree `dtree` is declared and allocated. The `dtree->train()` method is then called. In this case, the vector of `responses[]` (poisonous or edible) was set to the ASCII value of "p" or "e" (respectively) for each data point. After the `train()` method terminates, `dtree` is ready to be used for predicting new data. The decision tree may also be saved to disk via `save()` and loaded via `load()` (each method is shown below).* Between the saving and the loading, we reset and zero out the tree by calling the `clear()` method.

```
dtree->save("tree.xml","MyTree");

dtree->clear();

dtree->load("tree.xml","MyTree");
```

This saves and loads a tree file called *tree.xml*. (Using the *.xml* extension stores an XML data file; if we used a *.yml* or *.yaml* extension, it would store a YAML data file.) The optional "MyTree" is a tag that labels the tree within the *tree.xml* file. As with other statistical models in the machine learning module, multiple objects cannot be stored in a single *.xml* or *.yml* file when using `save()`; for multiple storage one needs to use `cvOpenFileStorage()` and `write()`. However, `load()` is a different story: this function can load an object by its name even if there is some other data stored in the file.

The function for prediction with a decision tree is:

```
CvDTreeNode* CvDTree::predict(
  const CvMat* _sample,
  const CvMat* _missing_data_mask = 0,
  bool         raw_mode           = false
) const;
```

Here `_sample` is a floating-point vector of features used to predict; `_missing_data_mask` is a byte vector of the same length and orientation† as the `_sample` vector, in which non-zero values indicate a missing feature value. Finally, `raw_mode` indicates unnormalized data with "false" (the default) or "true" for normalized input categorical data values. This is mainly used in ensembles of trees to speed up prediction. Normalizing data to fit within the (0, 1) interval is simply a computational speedup because the algorithm then knows the bounds in which data may fluctuate. Such normalization has no effect on accuracy. This method returns a node of the decision tree, and you may access the

* As mentioned previously, `save()` and `load()` are convenience wrappers for the more complex functions `write()` and `read()`.

† By "same . . . orientation" we mean that if the sample is a 1-by-*N* vector the mask must be 1-by-*N*, and if the sample is *N*-by-1 then the mask must be *N*-by-1.

predicted value using (CvDTreeNode *)->value which is returned by the dtree->predict() method (see CvDTree::predict() described previously):

```
double r = dtree->predict( &sample, &mask )->value;
```

Finally, we can call the useful var_importance() method to learn about the importance of the individual features. This function will return an *N*-by-1 vector of type double (CV_64FC1) containing each feature's relative importance for prediction, where the value 1 indicates the highest importance and 0 indicates absolutely not important or useful for prediction. Unimportant features may be eliminated on a second-pass training. (See Figure 13-12 for a display of variable importance.) The call is as follows:

```
const CvMat* var_importance = dtree->get_var_importance();
```

As demonstrated in the *…/opencv/samples/c/mushroom.cpp* file, individual elements of the importance vector may be accessed directly via

```
double val = var_importance->data.db[i];
```

Most users will only train and use the decision trees, but advanced or research users may sometimes wish to examine and/or modify the tree nodes or the splitting criteria. As stated in the beginning of this section, the information for how to do this is in the ML documentation that ships with OpenCV at *…/opencv/docs/ref/opencvref_ml.htm#ch_dtree*, which can also be accessed via the OpenCV Wiki (*http://opencvlibrary.sourceforge.net/*). The sections of interest for such advanced analysis are the class structure CvDTree{}, the training structure CvDTreeTrainData{}, the node structure CvDTreeNode{}, and its contained split structure CvDTreeSplit{}.

Decision Tree Results

Using the code just described, we can learn several things about edible or poisonous mushrooms from the *agaricus-lepiota.data* file. If we just train a decision tree without pruning, so that it learns the data perfectly, we get the tree shown in Figure 13-8. Although the full decision tree learns the training set of data perfectly, remember the lesson of Figure 13-2 (overfitting). What we've done in Figure 13-8 is to memorize the data together with its mistakes and noise. Thus, it is unlikely to perform well on real data. That is why OpenCV decision trees and CART type trees typically include an additional step of penalizing complex trees and pruning them back until complexity is in balance with performance. There are other decision tree implementations that grow the tree only until complexity is balanced with performance and so combine the pruning phase with the learning phase. However, during development of the ML library it was found that trees that are fully grown first and then pruned (as implemented in OpenCV) performed better than those that combine training with pruning in their generation phase.

Figure 13-9 shows a pruned tree that still does quite well (but not perfectly) on the training set but will probably perform better on real data because it has a better balance between bias and variance. Yet this classifier has an serious shortcoming: Although it performs well on the data, it still labels poisonous mushrooms as edible 1.23% of the time. Perhaps we'd be happier with a worse classifier that labeled many edible mushrooms as poisonous provided it never invited us to eat a poisonous mushroom! Such

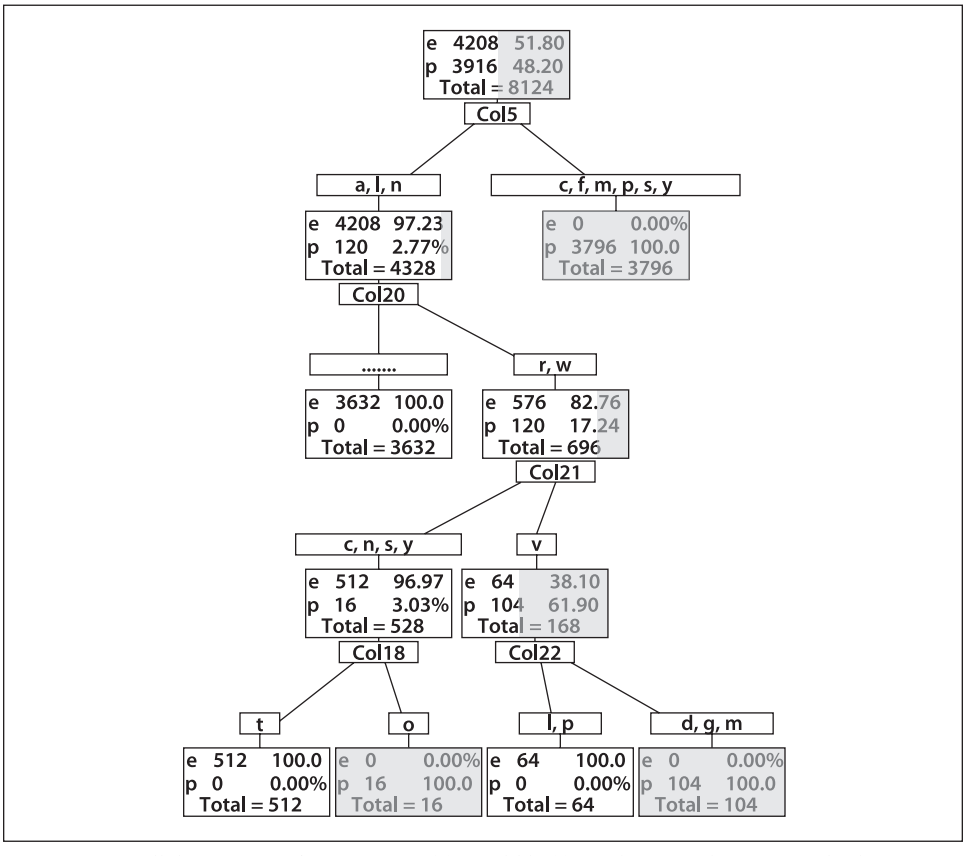

Figure 13-8. Full decision tree for poisonous (p) or edible (e) mushrooms: this tree was built out to full complexity for 0% error on the training set and so would probably suffer from variance problems on test or real data (the dark portion of a rectangle represents the poisonous portion of mushrooms at that phase of categorization)

a classifier can be created by intentionally biasing the classifier and/or the data. This is sometimes referred to as *adding a cost* to the classifier. In our case, we want to add a higher cost for misclassifying poisonous mushrooms than for misclassifying edible mushrooms. Cost can be imposed "inside" a classifier by changing the weighting of how much a "bad" data point counts versus a "good" one. OpenCV allows you to do this by adjusting the `priors` vector in the `CvDTreeParams{}` structure passed to the `train()` method, as we have discussed previously. Even without going inside the classifier code, we can impose a prior cost by duplicating (or resampling from) "bad" data. Duplicating "bad" data points implicitly gives a higher weight to the "bad" data, a technique that can work with any classifier.

Figure 13-10 shows a tree where a 10 × bias was imposed against poisonous mushrooms. This tree makes no mistakes on poisonous mushrooms at a cost of many more mistakes on edible mushrooms—a case of "better safe than sorry". Confusion matrices for the (pruned) unbiased and biased trees are shown in Figure 13-11.

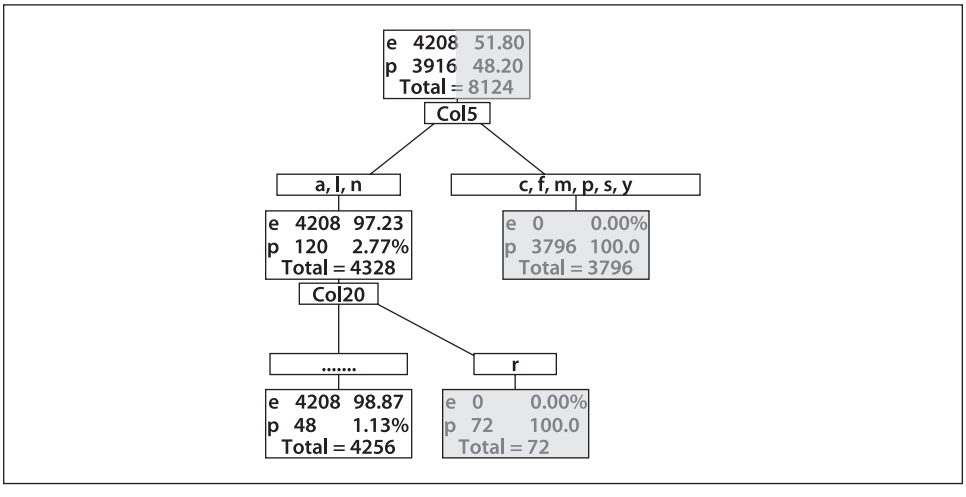

Figure 13-9. Pruned decision tree for poisonous (p) and edible (e) mushrooms: despite being pruned, this tree shows low error on the training set and would likely work well on real data

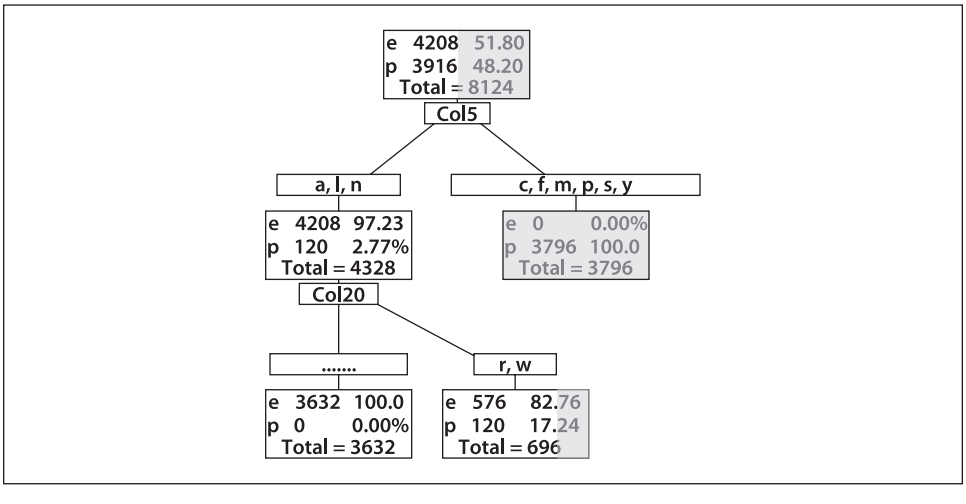

Figure 13-10. An edible mushroom decision tree with 10 × bias against misidentification of poisonous mushrooms as edible; note that the lower right rectangle, though containing a vast majority of edible mushrooms, does not contain a 10 × majority and so would be classified as inedible

Finally, we can learn something more from the data by using the *variable importance* machinery that comes with the tree-based classifiers in OpenCV.* Variable importance measurement techniques were discussed in a previous subsection, and they involve successively perturbing each feature and then measuring the effect on classifier performance. Features that cause larger drops in performance when perturbed are more important. Also, decision trees directly show importance via the splits they found in the

* Variable importance techniques may be used with any classifier, but at this time OpenCV implements them only with tree-based methods.

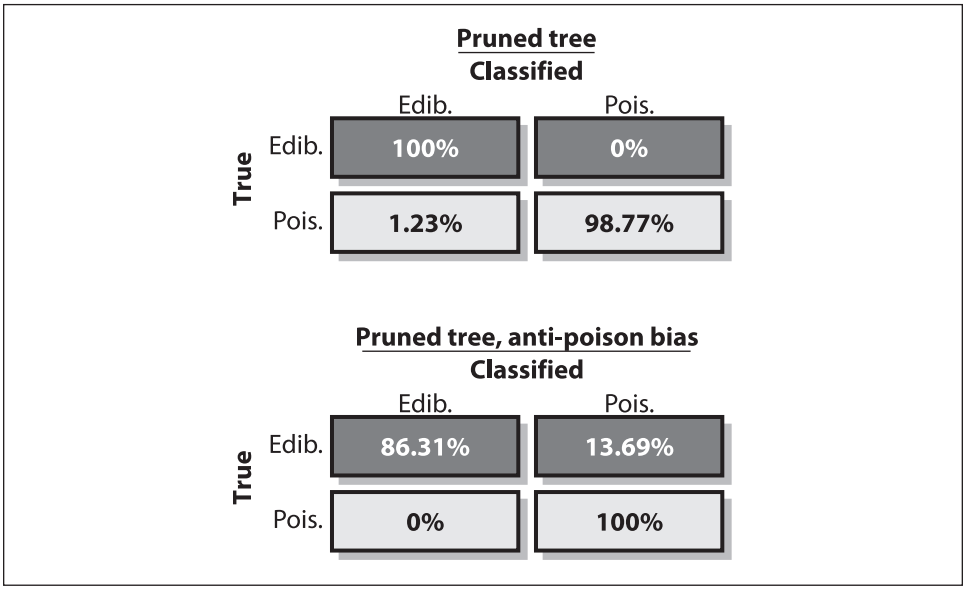

Figure 13-11. Confusion matrices for (pruned) edible mushroom decision trees: the unbiased tree yields better overall performance (top panel) but sometimes misclassifies poisonous mushrooms as edible; the biased tree does not perform as well overall (lower panel) but never misclassifies poisonous mushrooms

data: the first splits are presumably more important than later splits. Splits can be a useful indicator of importance, but they are done in a "greedy" fashion—finding which split most purifies the data *now*. It is often the case that doing a worse split first leads to better splits later, but these trees won't find this out.* The variable importance for poisonous mushrooms is shown in Figure 13-12 for both the unbiased and the biased trees. Note that the order of important variables changes depending on the bias of the trees.

Boosting

Decision trees are extremely useful, but they are often not the best-performing classifiers. In this and the next section we present two techniques, *boosting* and *random trees*, that use trees in their inner loop and so inherit many of the useful properties of trees (e.g., being able to deal with mixed and unnormalized data types and missing features). These techniques typically perform at or near the state of the art; thus they are often the best "out of the box" supervised classification techniques† available in the library.

Within in the field of supervised learning there is a *meta-learning* algorithm (first described by Michael Kerns in 1988) called *statistical boosting*. Kerns wondered whether

* OpenCV (following Breiman's technique) computes variable importance across all the splits, including surrogate ones, which decreases the possible negative effect that CART's greedy splitting algorithm would have on variable importance ratings.

† Recall that the "no free lunch" theorem informs us that there is no a priori "best" classifier. But on many data sets of interest in vision, boosting and random trees perform quite well.

Unbiased Importance:

Variable Name	Importance
Col5	100.00
Col20	58.39
Col9	40.74
Col12	37.31
Col13	34.90
Col19	32.30
Col8	30.94
Col4	26.69
Col15	21.21
Col14	21.15
Col21	20.89
Col22	17.67
Col7	12.88
Col18	4.44
Col2	4.21
Col3	3.77
Col1	2.89
Col10	1.10

Biased Importance:

Variable Name	Importance
Col5	100.00
Col15	23.86
Col14	22.81
Col21	19.47
Col20	19.42
Col7	15.10
Col18	11.90
Col9	11.11
Col11	9.48
Col1	8.99
Col10	8.55
Col3	8.45
Col22	8.37
Col12	5.30
Col17	5.26
Col8	5.08
Col13	3.41
Col6	2.30
Col19	1.37

Figure 13-12. Variable importance for edible mushroom as measured by an unbiased tree (left panel) and a tree biased against poison (right panel)

it is possible to learn a strong classifier out of many weak classifiers.* The first boosting algorithm, known as *AdaBoost*, was formulated shortly thereafter by Freund and Schapire.† OpenCV ships with four types of boosting:

- `CvBoost :: DISCRETE` (discrete AdaBoost)
- `CvBoost :: REAL` (real AdaBoost)
- `CvBoost :: LOGIT` (LogitBoost)
- `CvBoost :: GENTLE` (gentle AdaBoost)

Each of these are variants of the original AdaBoost, and often we find that the "real" and "gentle" forms of AdaBoost work best. *Real AdaBoost* is a technique that utilizes confidence-rated predictions and works well with categorical data. *Gentle AdaBoost* puts less weight on outlier data points and for that reason is often good with regression data. *LogitBoost* can also produce good regression fits. Because you need only set a flag, there's no reason not to try all types on a data set and then select the boosting method that works best.‡ Here we'll describe the original AdaBoost. For classification it should

* The output of a "weak classifier" is only weakly correlated with the true classifications, whereas that of a "strong classifier" is strongly correlated with true classifications. Thus, weak and strong are defined in a statistical sense.

† Y. Freund and R. E. Schapire, "Experiments with a New Boosting Algorithm", in *Machine Learning: Proceedings of the Thirteenth International Conference* (Morgan Kauman, San Francisco, 1996), 148–156.

‡ This procedure is an example of the machine learning metatechnique known as *voodoo learning* or *voodoo programming*. Although unprincipled, it is often an effective method of achieving the best possible performance. Sometimes, after careful thought, one can figure out why the best-performing method was the best, and this can lead to a deeper understanding of the data. Sometimes not.

be noted that, as implemented in OpenCV, boosting is a two-class (yes-or-no) classifier* (unlike the decision tree or random tree classifiers, which can handle multiple classes at once). Of the different OpenCV boosting methods, `LogitBoost` and `GentleBoost` (referenced in the "Boosting Code" subsection to follow) can be used to perform regression in addition to binary classification.

AdaBoost

Boosting algorithms are used to train T weak classifiers h_t, $t \in \{1,...,T\}$. These classifiers are generally very simple individually. In most cases these classifiers are decision trees with only one split (called *decision stumps*) or at most a few levels of splits (perhaps up to three). Each of the classifiers is assigned a weighted vote α_t in the final decision-making process. We use a labeled data set of input feature vectors x_i, each with scalar label y_i (where $i = 1,...,M$ data points). For AdaBoost the label is binary, $y_i \in \{-1,+1\}$, though it can be any floating-point number in other algorithms. We initialize a data point weighting distribution $D_t(i)$ that tells the algorithm how much misclassifying a data point will "cost". The key feature of boosting is that, as the algorithm progresses, this cost will evolve so that weak classifiers trained later will focus on the data points that the earlier trained weak classifiers tended to do poorly on. The algorithm is as follows.

1. $D_1(i) = 1/m$, $i = 1,...,m$.
2. For $t = 1,...,T$:
   a. Find the classifier h_t that minimizes the $D_t(i)$ weighted error:
   b. $h_t = \arg\min_{h_j \in H} \varepsilon_j$, where $\varepsilon_j = \sum_{i=1}^{m} D_t(i)$ (for $y_i \neq h_j(x_i)$) as long as $\varepsilon_j < 0.5$; else quit.
   c. Set the h_t voting weight $\alpha_t = \frac{1}{2}\log[(1-\varepsilon_t)/\varepsilon_t]$, where ε_t is the arg min error from step 2b.
   d. Update the data point weights: $D_{t+1}(i) = [D_t(i)\exp(-\alpha_t y_i h_t(x_i))]/Z_t$, where Z_t normalizes the equation over all data points i.

Note that, in step 2b, if we can't find a classifier with less than a 50% error rate then we quit; we probably need better features.

When the training algorithm just described is finished, the final strong classifier takes a new input vector x and classifies it using a weighted sum over the learned weak classifiers h_t:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

* There is a trick called *unrolling* that can be used to adapt any binary classifier (including boosting) for N-class classification problems, but this makes both training and prediction significantly more expensive. See *.../opencv/samples/c/letter_recog.cpp*.

Here, the sign function converts anything positive into a 1 and anything negative into a –1 (zero remains 0).

Boosting Code

There is example code in *…/opencv/samples/c/letter_recog.cpp* that shows how to use boosting, random trees and back-propagation (aka multilayer perception, MLP). The code for boosting is similar to the code for decision trees but with its own control parameters:

```
struct CvBoostParams : public CvDTreeParams {
  int    boost_type;        // CvBoost:: DISCRETE, REAL, LOGIT, GENTLE
  int    weak_count;        // How many classifiers
  int    split_criteria;    // CvBoost:: DEFAULT, GINI, MISCLASS, SQERR
  double weight_trim_rate;
  CvBoostParams();
  CvBoostParams(
    int          boost_type,
    int          weak_count,
    double       weight_trim_rate,
    int          max_depth,
    bool         use_surrogates,
    const float* priors
  );
};
```

In CvDTreeParams, boost_type selects one of the four boosting algorithms listed previously. The split_criteria is one of the following.

- CvBoost :: DEFAULT (use the default for the particular boosting method)
- CvBoost :: GINI (default option for real AdaBoost)
- CvBoost :: MISCLASS (default option for discrete AdaBoost)
- CvBoost :: SQERR (least-square error; only option available for LogitBoost and gentle AdaBoost)

The last parameter, weight_trim_rate, is for computational savings and is used as described next. As training goes on, many data points become unimportant. That is, the weight $D_t(i)$ for the ith data point becomes very small. The weight_trim_rate is a threshold between 0 and 1 (inclusive) that is implicitly used to throw away some training samples in a given boosting iteration. For example, suppose weight_trim_rate is set to 0.95. This means that samples with summary weight $\leq 1.0–0.95 = 0.05$ (5%) do not participate in the next iteration of training. Note the words "next iteration". The samples are not discarded forever. When the next weak classifier is trained, the weights are computed for all samples and so some previously insignificant samples may be returned back to the next training set. To turn this functionality off, set the weight_trim_rate value to 0.

Observe that CvBoostParams{} inherits from CvDTreeParams{}, so we may set other parameters that are related to decision trees. In particular, if we are dealing with features

that may be missing* then we can set use_surrogates to CvDTreeParams::use_surrogates, which will ensure that alternate features on which the splitting is based are stored at each node. An important option is that of using priors to set the "cost" of false positives. Again, if we are learning edible or poisonous mushrooms then we might set the priors to be float priors[] = {1.0, 10.0}; then each error of labeling a poisonous mushroom edible would cost ten times as much as labeling an edible mushroom poisonous.

The CvBoost class contains the member weak, which is a CvSeq* pointer to the weak classifiers that inherits from CvDTree decision trees.† For LogitBoost and GentleBoost, the trees are regression trees (trees that predict floating-point values); decision trees for the other methods return only votes for class 0 (if positive) or class 1 (if negative). This contained class sequence has the following prototype:

```
class CvBoostTree: public CvDTree {

public:
    CvBoostTree();
    virtual ~CvBoostTree();
    virtual bool train(
      CvDTreeTrainData* _train_data,
      const CvMat*      subsample_idx,
      CvBoost*          ensemble
    );
    virtual void scale( double s );
    virtual void read(
      CvFileStorage*    fs,
      CvFileNode*       node,
      CvBoost*          ensemble,
      CvDTreeTrainData* _data
    );
    virtual void clear();

protected:
    ...
    CvBoost* ensemble;

};
```

Training is almost the same as for decision trees, but there is an extra parameter called update that is set to false (0) by default. With this setting, we train a whole new ensemble of weak classifiers from scratch. If update is set to true (1) then we just add new weak classifiers onto the existing group. The function prototype for training a boosted classifier is:

* Note that, for computer vision, features are computed from an image and then fed to the classifier; hence they are almost never "missing". Missing features arise often in data collected by humans—for example, forgetting to take the patient's temperature one day.

† The naming of these objects is somewhat nonintuitive. The object of type CvBoost is the boosted tree classifier. The objects of type CvBoostTree are the weak classifiers that constitute the overall boosted strong classifier. Presumably, the weak classifiers are typed as CvBoostTree because they derive from CvDTree (i.e., they are little trees in themselves, albeit possibly so little that they are just stumps). The member variable weak of CvBoost points to a sequence enumerating the weak classifiers of type CvBoostTree.

```
bool CvBoost::train(
  const CvMat*  _train_data,
  int           _tflag,
  const CvMat*  _responses,
  const CvMat*  _var_idx       = 0,
  const CvMat*  _sample_idx    = 0,
  const CvMat*  _var_type      = 0,
  const CvMat*  _missing_mask = 0,
  CvBoostParams params         = CvBoostParams(),
  bool          update         = false
);
```

An example of training a boosted classifier may be found in *…/opencv/samples/c/letter_recog.cpp*. The training code snippet is shown in Example 13-3.

Example 13-3. Training snippet for boosted classifiers

```
var_type = cvCreateMat( var_count + 2, 1, CV_8U );

cvSet( var_type, cvScalarAll(CV_VAR_ORDERED) );

// the last indicator variable, as well
// as the new (binary) response are categorical
//
cvSetReal1D( var_type, var_count, CV_VAR_CATEGORICAL );
cvSetReal1D( var_type, var_count+1, CV_VAR_CATEGORICAL );

// Train the classifier
//
boost.train(
  new_data,
  CV_ROW_SAMPLE,
  responses,
  0,
  0,
  var_type,
  0,
  CvBoostParams( CvBoost::REAL, 100, 0.95, 5, false, 0 )
);

cvReleaseMat( &new_data );
cvReleaseMat( &new_responses );
```

The prediction function for boosting is also similar to that for decision trees:

```
float CvBoost::predict(
  const CvMat* sample,
  const CvMat* missing        = 0,
  CvMat*       weak_responses = 0,
  CvSlice      slice          = CV_WHOLE_SEQ,
  bool         raw_mode       = false
) const;
```

To perform a simple prediction, we pass in the feature vector `sample` and then `predict()` returns the predicted value. Of course, there are a variety of optional parameters. The first of these is the `missing` feature mask, which is the same as it was for decision trees;

it consists of a byte vector of the same dimension as the sample vector, where nonzero values indicate a missing feature. (Note that this mask cannot be used unless you have trained the classifier with the use_surrogates parameter set to CvDTreeParams::use_surrogates.)

If we want to get back the responses of each of the weak classifiers, we can pass in a floating-point CvMat vector, weak_responses, with length equal to the number of weak classifiers. If weak_responses is passed, CvBoost::predict will fill the vector with the response of each individual classifier:

```
CvMat* weak_responses = cvCreateMat(
  1,
  boostedClassifier.get_weak_predictors()->total,
  CV_32F
);
```

The next prediction parameter, slice, indicates which contiguous subset of the weak classifiers to use; it can be set by

```
inline CvSlice cvSlice( int start, int end );
```

However, we usually just accept the default and leave slice set to "every weak classifier" (CvSlice slice=CV_WHOLE_SEQ). Finally, we have the raw_mode, which is off by default but can be turned on by setting it to true. This parameter is exactly the same as for decision trees and indicates that the data is prenormalized to save computation time. Normally you won't need to use this. An example call for boosted prediction is

```
boost.predict( temp_sample, 0, weak_responses );
```

Finally, some auxiliary functions may be of use from time to time. We can remove a weak classifier from the learned model via

```
void CvBoost::prune( CvSlice slice );
```

We can also return all the weak classifiers for examination:

```
CvSeq* CvBoost::get_weak_predictors();
```

This function returns a CvSeq of pointers to CvBoostTree.

Random Trees

OpenCV contains a *random trees* class, which is implemented following Leo Breiman's theory of *random forests*.* Random trees can learn more than one class at a time simply by collecting the class "votes" at the leaves of each of many trees and selecting the class receiving the maximum votes as the winner. Regression is done by averaging the values across the leaves of the "forest". Random trees consist of randomly perturbed decision trees and are among the best-performing classifiers on data sets studied while the ML library was being assembled. Random trees also have the potential for parallel implementation, even on nonshared memory systems, a feature that lends itself to increased use in the future. The basic subsystem on which random trees are built is once again a decision tree. This decision tree is built all the way down until it's *pure*. Thus (cf. the upper right

* Most of Breiman's work on random forests is conveniently collected on a single website (*http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm*).

panel of Figure 13-2), each tree is a high-variance classifier that nearly perfectly learns its training data. To counterbalance the high variance, we average together many such trees (hence the name random trees).

Of course, averaging trees will do us no good if the trees are all very similar to each other. To overcome this, random trees cause each tree to be different by randomly selecting a different feature subset of the total features from which the tree may learn at each node. For example, an object-recognition tree might have a long list of potential features: color, texture, gradient magnitude, gradient direction, variance, ratios of values, and so on. Each node of the tree is allowed to choose from a random subset of these features when determining how best to split the data, and each subsequent node of the tree gets a new, randomly chosen subset of features on which to split. The size of these random subsets is often chosen as the square root of the number of features. Thus, if we had 100 potential features then each node would randomly choose 10 of the features and find a best split of the data from among those 10 features. To increase robustness, random trees use an *out of bag* measure to verify splits. That is, at any given node, training occurs on a new subset of the data that is randomly selected *with replacement*,* and the rest of the data—those values not randomly selected, called "out of bag" (or OOB) data—are used to estimate the performance of the split. The OOB data is usually set to have about one third of all the data points.

Like all tree-based methods, random trees inherit many of the good properties of trees: surrogate splits for missing values, handling of categorical and numerical values, no need to normalize values, and easy methods for finding variables that are important for prediction. Random trees also used the OOB error results to estimate how well it will do on unseen data. If the training data has a similar distribution to the test data, this OOB performance prediction can be quite accurate.

Finally, random trees can be used to determine, for any two data points, their *proximity* (which in this context means "how alike" they are, not "how near" they are). The algorithm does this by (1) "dropping" the data points into the trees, (2) counting how many times they end up in the same leaf, and (3) dividing this "same leaf" count by the total number of trees. A proximity result of 1 is exactly similar and 0 means very dissimilar. This proximity measure can be used to identify outliers (those points very unlike any other) and also to cluster points (group close points together).

Random Tree Code

We are by now familiar with how the ML library works, and random trees are no exception. It starts with a parameter structure, `CvRTParams`, which it inherits from decision trees:

```
struct CvRTParams : public CvDTreeParams {

  bool          calc_var_importance;
  int           nactive_vars;
```

* This means that some data points might be randomly repeated.

```
  CvTermCriteria term_crit;

  CvRTParams() : CvDTreeParams(
    5, 10, 0, false,
    10, 0, false, false,
    0
  ), calc_var_importance(false), nactive_vars(0) {

    term_crit = cvTermCriteria(
      CV_TERMCRIT_ITER | CV_TERMCRIT_EPS,
      50,
      0.1
    );
  }

  CvRTParams(
    int           _max_depth,
    int           _min_sample_count,
    float         _regression_accuracy,
    bool          _use_surrogates,
    int           _max_categories,
    const float* _priors,
    bool          _calc_var_importance,
    int           _nactive_vars,
    int           max_tree_count,
    float         forest_accuracy,
    int           termcrit_type,
  );

};
```

The key new parameters in `CvRTParams` are `calc_var_importance`, which is just a switch to calculate the variable importance of each feature during training (at a slight cost in additional computation time). Figure 13-13 shows the variable importance computed on a subset of the mushroom data set that ships with OpenCV in the *…/opencv/samples/c/agaricus-lepiota.data* file. The `nactive_vars` parameter sets the size of the randomly selected subset of features to be tested at any given node and is typically set to the square root of the total number of features; `term_crit` (a structure discussed elsewhere in this chapter) is the control on the maximum number of trees. For learning random trees, in `term_crit` the `max_iter` parameter sets the total number of trees; `epsilon` sets the "stop learning" criteria to cease adding new trees when the error drops below the OOB error; and the `type` tells which of the two stopping criteria to use (usually it's both: `CV_TERMCRIT_ITER | CV_TERMCRIT_EPS`).

Random trees training has the same form as decision trees training (see the deconstruction of `CvDTree::train()` in the subsection on "Training the Tree") except that is uses the `CvRTParam` structure:

```
bool CvRTrees::train(
  const CvMat* train_data,
  int          tflag,
  const CvMat* responses,
  const CvMat* comp_idx     = 0,
```

```
  const CvMat* sample_idx   = 0,
  const CvMat* var_type     = 0,
  const CvMat* missing_mask = 0,
  CvRTParams   params       = CvRTParams()
);
```

Variable Name	RandomTrees	Boosting	DecisionTree
Col5	100.00	100.00	100.00
Col20	35.20	58.89	57.37
Col21	16.47	6.11	34.51
Col19	13.35	4.57	26.11
Col9	13.01	43.15	45.96
Col13	10.02	24.47	26.85
Col8	9.52	37.51	42.28
Col12	9.09	27.66	28.90
Col22	8.29	0.28	20.00
Col7	6.08	0.10	21.33
Col15	4.06	1.84	21.41
Col11	3.52	0.44	16.29
Col4	3.12		14.67
Col14	2.98	0.25	20.81
Col18	2.68		0.70
Col3	2.56	0.11	9.15
Col2	2.22	0.39	12.14
Col10	1.79		2.67
Col1	0.41	0.24	7.26
Col17	0.18	0.32	0.54
Col0			
Col6			
Col16			

Figure 13-13. Variable importance over the mushroom data set for random trees, boosting, and decision trees: random trees used fewer significant variables and achieved the best prediction (100% correct on a randomly selected test set covering 20% of data)

An example of calling the train function for a multiclass learning problem is provided in the samples directory that ships with OpenCV; see the *…/opencv/samples/c/letter_recog.cpp* file, where the random trees classifier is named `forest`.

```
forest.train(
  data,
  CV_ROW_SAMPLE,
  responses,
  0,
  sample_idx,
  var_type,
  0,
  CvRTParams(10,10,0,false,15,0,true,4,100,0.01f,CV_TERMCRIT_ITER)
);
```

Random trees prediction has a form similar to that of the decision trees prediction function `CvDTree::predict`, but rather than return a `CvDTreeNode*` pointer it returns the

average return value over all the trees in the forest. The missing mask is an optional parameter of the same dimension as the sample vector, where nonzero values indicate a missing feature value in sample.

```
double CvRTrees::predict(
  const CvMat* sample,
  const CvMat* missing = 0
) const;
```

An example prediction call from the *letter_recog.cpp* file is

```
double r;
CvMat  sample;

cvGetRow( data, &sample, i );

r = forest.predict( &sample );
r = fabs((double)r - responses->data.fl[i]) <= FLT_EPSILON ? 1 : 0;
```

In this code, the return variable r is converted into a count of correct predictions.

Finally, there are random tree analysis and utility functions. Assuming that CvRTParams::calc_var_importance is set in training, we can obtain the relative importance of each variable by

```
const CvMat* CvRTrees::get_var_importance() const;
```

See Figure 13-13 for an example of variable importance for the mushroom data set from random trees. We can also obtain a measure of the learned random trees model proximity of one data point to another by using the call

```
float CvRTrees::get_proximity(
  const CvMat* sample_1,
  const CvMat* sample_2
) const;
```

As mentioned previously, the returned proximity is 1 if the data points are identical and 0 if the points are completely different. This value is usually between 0 and 1 for two data points drawn from a distribution similar to that of the training set data.

Two other useful functions give the total number of trees or the data structure containing a given decision tree:

```
int           get_tree_count() const; // How many trees are in the forest
CvForestTree* get_tree(int i) const;  // Get an individual decision tree
```

Using Random Trees

We've remarked that the random trees algorithm often performs the best (or among the best) on the data sets we tested, but the best policy is still to try many classifiers once you have your training data defined. We ran random trees, boosting, and decision trees on the mushroom data set. From the 8,124 data points we randomly extracted 1,624 test points, leaving the remainder as the training set. After training these three tree-based classifiers with their default parameters, we obtained the results shown in Table 13-4 on the test set. The mushroom data set is fairly easy and so—although random trees did the

best—it wasn't such an overwhelming favorite that we can definitively say which of the three classifiers works better on this particular data set.

Table 13-4. Results of tree-based methods on the OpenCV mushroom data set (1,624 randomly chosen test points with no extra penalties for misclassifying poisonous mushrooms)

Classifier	Performance Results
Random trees	100%
AdaBoost	99%
Decision trees	98%

What is more interesting is the variable importance (which we also measured from the classifiers), shown in Figure 13-13. The figure shows that random trees and boosting each used significantly fewer important variables than required by decision trees. Above 15% significance, random trees used only three variables and boosting used six whereas decision trees needed thirteen. We could thus shrink the feature set size to save computation and memory and still obtain good results. Of course, for the decision trees algorithm you have just a single tree while for random trees and AdaBoost you must evaluate multiple trees; thus, which method has the least computational cost depends on the nature of the data being used.

Face Detection or Haar Classifier

We now turn to the final tree-based technique in OpenCV: the *Haar classifier*, which builds a *boosted rejection cascade*. It has a different format from the rest of the ML library in OpenCV because it was developed earlier as a full-fledged face-recognition application. Thus, we cover it in detail and show how it can be trained to recognize faces and other rigid objects.

Computer vision is a broad and fast-changing field, so the parts of OpenCV that implement a specific technique—rather than a component algorithmic piece—are more at risk of becoming out of date. The face detector that comes with OpenCV is in this "risk" category. However, face detection is such a common need that it is worth having a baseline technique that works fairly well; also, the technique is built on the well-known and often used field of statistical boosting and thus is of more general use as well. In fact, several companies have engineered the "face" detector in OpenCV to detect "mostly rigid" objects (faces, cars, bikes, human body) by training new detectors on many thousands of selected training images for each view of the object. This technique has been used to create state-of-the-art detectors, although with a different detector trained for each view or pose of the object. Thus, the Haar classifier is a valuable tool to keep in mind for such recognition tasks.

OpenCV implements a version of the face-detection technique first developed by Paul Viola and Michael Jones—commonly known as the *Viola-Jones detector**—and later

* P. Viola and M. J. Jones, "Rapid Object Detection Using a Boosted Cascade of Simple Features," *IEEE CVPR* (2001).

extended by Rainer Lienhart and Jochen Maydt* to use *diagonal features* (more on this distinction to follow). OpenCV refers to this detector as the "Haar classifier" because it uses Haar features† or, more precisely, Haar-like wavelets that consist of adding and subtracting rectangular image regions before thresholding the result. OpenCV ships with a set of pretrained object-recognition files, but the code also allows you to train and store new object models for the detector. We note once again that the training (`createsamples()`, `haartraining()`) and detecting (`cvHaarDetectObjects()`) code works well on any objects (not just faces) that are consistently textured and mostly rigid.

The pretrained objects that come with OpenCV for this detector are in *…/opencv/data/haarcascades*, where the model that works best for frontal face detection is *haarcascade_frontalface_alt2.xml*. Side face views are harder to detect accurately with this technique (as we shall describe shortly), and those shipped models work less well. If you end up training good object models, perhaps you will consider contributing them as open source back to the community.

Supervised Learning and Boosting Theory

The Haar classifier that is included in OpenCV is a supervised classifier (these were discussed at the beginning of the chapter). In this case we typically present histogram- and size-equalized image patches to the classifier, which are then labeled as containing (or not containing) the object of interest, which for this classifier is most commonly a face.

The Viola-Jones detector uses a form of AdaBoost but organizes it as a *rejection cascade* of nodes, where each node is a multitree AdaBoosted classifier designed to have high (say, 99.9%) detection rate (low false negatives, or missed faces) at the cost of a low (near 50%) rejection rate (high false positives, or "nonfaces" wrongly classified). For each node, a "not in class" result at any stage of the cascade terminates the computation, and the algorithm then declares that no face exists at that location. Thus, true class detection is declared only if the computation makes it through the entire cascade. For instances where the true class is rare (e.g., a face in a picture), rejection cascades can greatly reduce total computation because most of the regions being searched for a face terminate quickly in a nonclass decision.

Boosting in the Haar cascade

Boosted classifiers were discussed earlier in this chapter. For the Viola-Jones rejection cascade, the weak classifiers that it boosts in each node are decision trees that often are only one level deep (i.e., "decision stumps"). A decision stump is allowed just one decision of the following form: "Is the value v of a particular feature f above or below some threshold t"; then, for example, a "yes" indicates face and a "no" indicates no face:

* R. Lienhart and J. Maydt, "An Extended Set of Haar-like Features for Rapid Object Detection," *IEEE ICIP* (2002), 900–903.

† This is technically not correct. The classifier uses the threshold of the sums and differences of rectangular regions of data produced by any feature detector, which may include the Haar case of rectangles of raw (grayscale) image values. Henceforth we will use the term "Haar-like" in deference to this distinction.

$$f_i = \begin{cases} +1 & v_i \geq t_i \\ -1 & v_i < t_i \end{cases}$$

The number of Haar-like features that the Viola-Jones classifier uses in each weak classifier can be set in training, but mostly we use a single feature (i.e., a tree with a single split) or at most about three features. Boosting then iteratively builds up a classifier as a weighted sum of these kinds of weak classifiers. The Viola-Jones classifier uses the classification function:

$$F = \text{sign}(w_1 f_1 + w_2 f_2 + \cdots + w_n f_n)$$

Here, the sign function returns –1 if the number is less than 0, 0 if the number equals 0, and +1 if the number is positive. On the first pass through the data set, we learn the threshold t_1 of f_1 that best classifies the input. Boosting then uses the resulting errors to calculate the weighted vote w_1. As in traditional AdaBoost, each feature vector (data point) is also reweighted low or high according to whether it was classified correctly or not* in that iteration of the classifier. Once a node is learned this way, the surviving data from higher up in the cascade is used to train the next node and so on.

Viola-Jones Classifier Theory

The Viola-Jones classifier employs AdaBoost at each node in the cascade to learn a high detection rate at the cost of low rejection rate multitree (mostly multistump) classifier at each node of the cascade. This algorithm incorporates several innovative features.

1. It uses Haar-like input features: a threshold applied to sums and differences of rectangular image regions.
2. Its *integral image* technique enables rapid computation of the value of rectangular regions or such regions rotated 45 degrees (see Chapter 6). This data structure is used to accelerate computation of the Haar-like input features.
3. It uses statistical boosting to create binary (face–not face) classification nodes characterized by high detection and weak rejection.
4. It organizes the weak classifier nodes of a rejection cascade. In other words: the first group of classifiers is selected that best detects image regions containing an object while allowing many mistaken detections; the next classifier group† is the second-best at detection with weak rejection; and so forth. In test mode, an object is detected only if it makes it through the entire cascade.‡

* There is sometimes confusion about boosting lowering the classification weight on points it classifies correctly in training and raising the weight on points it classified wrongly. The reason is that boosting attempts to focus on correcting the points that it has "trouble" on and to ignore points that it already "knows" how to classify. One of the technical terms for this is that boosting is a margin maximize.

† Remember that each "node" in a rejection cascade is an AdaBoosted group of classifiers.

‡ This allows the cascade to run quickly, because it almost immediately rejects image regions that don't contain the object (and hence need not process through the rest of the cascade).

The Haar-like features used by the classifier are shown in Figure 13-14. At all scales, these features form the "raw material" that will be used by the boosted classifiers. They are rapidly computed from the integral image (see Chapter 6) representing the original grayscale image.

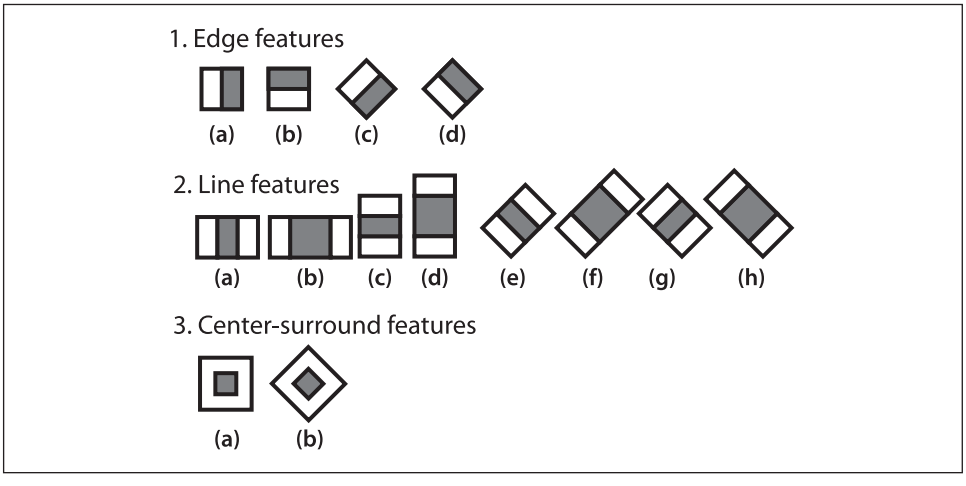

Figure 13-14. Haar-like features from the OpenCV source distribution (the rectangular and rotated regions are easily calculated from the integral image): in this diagrammatic representation of the wavelets, the light region is interpreted as "add that area" and the dark region as "subtract that area"

Viola and Jones organized each boosted classifier group into nodes of a rejection cascade, as shown in Figure 13-15. In the figure, each of the nodes F_j contains an entire boosted cascade of groups of decision stumps (or trees) trained on the Haar-like features from faces and nonfaces (or other objects the user has chosen to train on). Typically, the nodes are ordered from least to most complex so that computations are minimized (simple nodes are tried first) when rejecting easy regions of the image. Typically, the boosting in each node is tuned to have a very high detection rate (at the usual cost of many false positives). When training on faces, for example, almost all (99.9%) of the faces are found but many (about 50%) of the nonfaces are erroneously "detected" at each node. But this is OK because using (say) 20 nodes will still yield a face detection rate (through the whole cascade) of $0.999^{20} \approx 98\%$ with a false positive rate of only $0.5^{20} \approx 0.0001\%$!

During the run mode, a search region of different sizes is swept over the original image. In practice, 70–80% of nonfaces are rejected in the first two nodes of the rejection cascade, where each node uses about ten decision stumps. This quick and early "attentional reject" vastly speeds up face detection.

Works well on . . .

This technique implements face detection but is not limited to faces; it also works fairly well on other (mostly rigid) objects that have distinguishing views. That is, front views

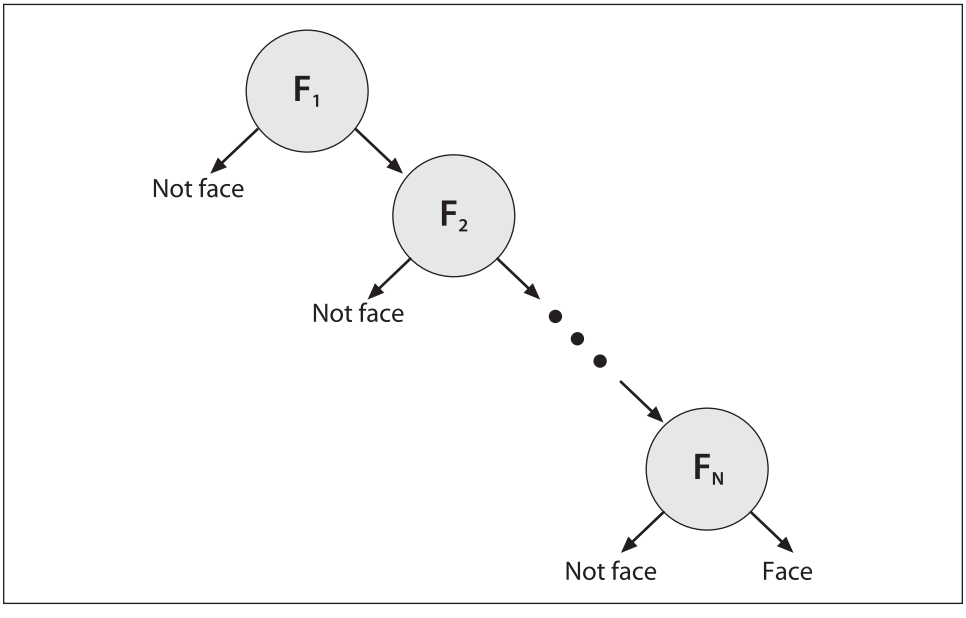

Figure 13-15. Rejection cascade used in the Viola-Jones classifier: each node represents a multitree boosted classifier ensemble tuned to rarely miss a true face while rejecting a possibly small fraction of nonfaces; however, almost all nonfaces have been rejected by the last node, leaving only true faces

of faces work well; backs, sides, or fronts of cars work well; but side views of faces or "corner" views of cars work less well—mainly because these views introduce variations in the template that the "blocky" features (see next paragraph) used in this detector cannot handle well. For example, a side view of a face must catch part of the changing background in its learned model in order to include the profile curve. To detect side views of faces, you may try *haarcascade_profileface.xml*, but to do a better job you should really collect much more data than this model was trained with and perhaps expand the data with different backgrounds behind the face profiles. Again, profile views are hard for this classifier because it uses block features and so is forced to attempt to learn the background variability that "peaks" through the informative profile edge of the side view of faces. In training, it's more efficient to learn only (say) right profile views. Then the test procedure would be to (1) run the right-profile detector and then (2) flip the image on its vertical axis and run the right-profile detector again to detect left-facing profiles.

As we have discussed, detectors based on these Haar-like features work well with "blocky" features—such as eyes, mouth, and hairline—but work less well with tree branches, for example, or when the object's outline shape is its most distinguishing characteristic (as with a coffee mug).

All that being said, if you are willing to gather lots of good, well-segmented data on fairly rigid objects, then this classifier can still compete with the best, and its construction as a rejection cascade makes it very fast to run (though not to train, however). Here "lots of data" means thousands of object examples and tens of thousands of nonobject examples.

By “good” data we mean that one shouldn’t mix, for instance, tilted faces with upright faces; instead, keep the data divided and use two classifiers, one for tilted and one for upright. “Well-segmented” data means data that is consistently boxed. Sloppiness in box boundaries of the training data will often lead the classifier to correct for fictitious variability in the data. For example, different placement of the eye locations in the face data location boxes can lead the classifier to assume that eye locations are not a geometrically fixed feature of the face and so can move around. Performance is almost always worse when a classifier attempts to adjust to things that aren’t actually in the real data.

Code for Detecting Faces

The `detect_and_draw()` code shown in Example 13-4 will detect faces and draw their found locations in different-colored rectangles on the image. As shown in the fourth through seventh (comment) lines, this code presumes that a previously trained classifier cascade has been loaded and that memory for detected faces has been created.

Example 13-4. Code for detecting and drawing faces

```
// Detect and draw detected object boxes on image
// Presumes 2 Globals:
//  Cascade is loaded by:
//     cascade = (CvHaarClassifierCascade*)cvLoad( cascade_name,
//   0, 0, 0 );
//  AND that storage is allocated:
//  CvMemStorage* storage = cvCreateMemStorage(0);
//
void detect_and_draw(
  IplImage* img,
  Double    scale = 1.3
){
  static CvScalar colors[] = {
    {{0,0,255}},   {{0,128,255}},{{0,255,255}},{{0,255,0}},
    {{255,128,0}},{{255,255,0}},{{255,0,0}},   {{255,0,255}}
  }; //Just some pretty colors to draw with

  // IMAGE PREPARATION:
  //
  IplImage* gray = cvCreateImage( cvSize(img->width,img->height), 8, 1 );
  IplImage* small_img = cvCreateImage(
    cvSize( cvRound(img->width/scale), cvRound(img->height/scale)), 8, 1
  );
  cvCvtColor( img, gray, CV_BGR2GRAY );
  cvResize( gray, small_img, CV_INTER_LINEAR );
  cvEqualizeHist( small_img, small_img );

  // DETECT OBJECTS IF ANY
  //
  cvClearMemStorage( storage );
  CvSeq* objects = cvHaarDetectObjects(
    small_img,
    cascade,
    storage,
```

Example 13-4. Code for detecting and drawing faces (continued)

```
    1.1,
    2,
    0 /*CV_HAAR_DO_CANNY_PRUNING*/,
    cvSize(30, 30)
  );

  // LOOP THROUGH FOUND OBJECTS AND DRAW BOXES AROUND THEM
  //
  for(int i = 0; i < (objects ? objects->total : 0); i++ ) {
    CvRect* r = (CvRect*)cvGetSeqElem( objects, i );
    cvRectangle(
      img,
      cvPoint(r.x,r.y),
      cvPoint(r.x+r.width,r.y+r.height),
      colors[i%8]
    )
  }
  cvReleaseImage( &graygray );
  cvReleaseImage( &small_img );
}
```

For convenience, in this code the `detect_and_draw()` function has a static array of color vectors `colors[]` that can be indexed to draw found faces in different colors. The classifier works on grayscale images, so the color BGR image `img` passed into the function is converted to grayscale using `cvCvtColor()` and then optionally resized in `cvResize()`. This is followed by histogram equalization via `cvEqualizeHist()`, which spreads out the brightness values—necessary because the integral image features are based on differences of rectangle regions and, if the histogram is not balanced, these differences might be skewed by overall lighting or exposure of the test images. Since the classifier returns found object rectangles as a sequence object `CvSeq`, we need to clear the global storage that we're using for these returns by calling `cvClearMemStorage()`. The actual detection takes place just above the `for{}` loop, whose parameters are discussed in more detail below. This loop steps through the found face rectangle regions and draws them in different colors using `cvRectangle()`. Let us take a closer look at detection function call:

```
CvSeq* cvHaarDetectObjects(
   const CvArr*             image,
   CvHaarClassifierCascade* cascade,
   CvMemStorage*            storage,
   double                   scale_factor  = 1.1,
   int                      min_neighbors = 3,
   int                      flags         = 0,
   CvSize                   min_size      = cvSize(0,0)
);
```

`CvArr image` is a grayscale image. If region of interest (ROI) is set, then the function will respect that region. Thus, one way of speeding up face detection is to trim down the image boundaries using ROI. The classifier cascade is just the Haar feature cascade that we loaded with `cvLoad()` in the face detect code. The `storage` argument is an OpenCV "work buffer" for the algorithm; it is allocated with `cvCreateMemStorage(0)` in the face detection

code and cleared for reuse with `cvClearMemStorage(storage)`. The `cvHaarDetectObjects()` function scans the input image for faces at all scales. Setting the `scale_factor` parameter determines how big of a jump there is between each scale; setting this to a higher value means faster computation time at the cost of possible missed detections if the scaling misses faces of certain sizes. The `min_neighbors` parameter is a control for preventing false detection. Actual face locations in an image tend to get multiple "hits" in the same area because the surrounding pixels and scales often indicate a face. Setting this to the default (3) in the face detection code indicates that we will only decide a face is present in a location if there are at least three overlapping detections. The `flags` parameter has four valid settings, which (as usual) may be combined with the Boolean OR operator. The first is `CV_HAAR_DO_CANNY_PRUNING`. Setting `flags` to this value causes flat regions (no lines) to be skipped by the classifier. The second possible flag is `CV_HAAR_SCALE_IMAGE`, which tells the algorithm to scale the image rather than the detector (this can yield some performance advantages in terms of how memory and cache are used). The next flag option, `CV_HAAR_FIND_BIGGEST_OBJECT`, tells OpenCV to return only the largest object found (hence the number of objects returned will be either one or none).* The final flag is `CV_HAAR_DO_ROUGH_SEARCH`, which is used only with `CV_HAAR_FIND_BIGGEST_OBJECT`. This flag is used to terminate the search at whatever scale the first candidate is found (with enough neighbors to be considered a "hit"). The final parameter, `min_size`, is the smallest region in which to search for a face. Setting this to a larger value will reduce computation at the cost of missing small faces. Figure 13-16 shows results for using the face-detection code on a scene with faces.

Learning New Objects

We've seen how to load and run a previously trained classifier cascade stored in an XML file. We used the `cvLoad()` function to load it and then used `cvHaarDetectObjects()` to find objects similar to the ones it was trained on. We now turn to the question of how to train our own classifiers to detect other objects such as eyes, walking people, cars, et cetera. We do this with the OpenCV *haartraining* application, which creates a classifier given a training set of positive and negative samples. The four steps of training a classifier are described next. (For more details, see the *haartraining* reference manual supplied with OpenCV in the *opencv/apps/HaarTraining/doc* directory.)

1. Gather a data set consisting of examples of the object you want to learn (e.g., front views of faces, side views of cars). These may be stored in one or more directories indexed by a text file in the following format:

   ```
   <path>/img_name_1 count_1 x11 y11 w11 h11 x12 y12 . . .
   <path>/img_name_2 count_2 x21 y21 w21 h21 x22 y22 . . .
      . . .
   ```

   Each of these lines contains the path (if any) and file name of the image containing the object(s). This is followed by the count of how many objects are in that image and then

* It is best not to use `CV_HAAR_DO_CANNY_PRUNING` with `CV_HAAR_FIND_BIGGEST_OBJECT`. Using both will seldom yield a performance gain; in fact, the net effect will often be a performance loss.

Figure 13-16. Face detection on a park scene: some tilted faces are not detected, and there is also a false positive (shirt near the center); for the 1054-by-851 image shown, more than a million sites and scales were searched to achieve this result in about 1.5 seconds on a 2 GHz machine

a list of rectangles containing the objects. The format of the rectangles is the *x*- and *y*-coordinates of the upper left corner followed by the width and height in pixels.

To be more specific, if we had a data set of faces located in directory *data/faces/*, then the index file *faces.idx* might look like this:

```
data/faces/face_000.jpg 2 73 100 25 37 133 123 30 45
data/faces/face_001.jpg 1 155 200 55 78
   . . .
```

If you want your classifier to work well, you will need to gather a lot of high-quality data (1,000–10,000 positive examples). "High quality" means that you've removed all unnecessary variance from the data. For example, if you are learning faces, you should align the eyes (and preferably the nose and mouth) as much as possible. The intuition here is that otherwise you are teaching the classifier that eyes need not appear at fixed locations in the face but instead could be anywhere within some region. Since this is not true of real data, your classifier will not perform as well. One strategy is to first train a cascade on a subpart, say "eyes", which are easier to align. Then use eye detection to find the eyes and rotate/resize the face until the eyes are

aligned. For asymmetric data, the "trick" of flipping an image on its vertical axis was described previously in the subsection "Works well on . . .".

2. Use the utility application `createsamples` to build a vector output file of the positive samples. Using this file, you can repeat the training procedure below on many runs, trying different parameters while using the same vector output file. For example:

   ```
   createsamples -vec faces.vec -info faces.idx -w 30 -h 40
   ```

   This reads in the *faces.idx* file described in step 1 and outputs a formatted training file, *faces.vec*. Then `createsamples` extracts the positive samples from the images before normalizing and resizing them to the specified width and height (here, 30-by-40). Note that `createsamples` can also be used to synthesize data by applying geometric transformations, adding noise, altering colors, and so on. This procedure could be used (say) to learn a corporate logo, where you take just one image and put it through various distortions that might appear in real imagery. More details can be found in the OpenCV reference manual *haartraining* located in */apps/HaarTraining/doc/*.

3. The Viola-Jones cascade is a binary classifier: It simply decides whether or not ("yes" or "no") the object in an image is similar to the training set. We've described how to collect and process the "yes" samples that contained the object of choice. Now we turn to describing how to collect and process the "no" samples so that the classifier can learn what does *not* look like our object. Any image that doesn't contain the object of interest can be turned into a negative sample. It is best to take the "no" images from the same type of data we will test on. That is, if we want to learn faces in online videos, for best results we should take our negative samples from comparable frames (i.e., other frames from the same video). However, respectable results can still be achieved using negative samples taken from just about anywhere (e.g., CD or Internet image collections). Again we put the images into one or more directories and then make an index file consisting of a list of image filenames, one per line. For example, an image index file called *backgrounds.idx* might contain the following path and filenames of image collections:

   ```
   data/vacations/beach.jpg
   data/nonfaces/img_043.bmp
   data/nonfaces/257-5799_IMG.JPG
      . . .
   ```

4. *Training*. Here's an example training call that you could type on a command line or create using a batch file:

   ```
   Haartraining /
     -data face_classifier_take_3 /
     -vec faces.vec -w 30 -h 40 /
     -bg backgrounds.idx /
     -nstages 20 /
     -nsplits 1 /
     [-nonsym] /
     -minhitrate 0.998 /
     -maxfalsealarm 0.5
   ```

In this call the resulting classifier will be stored in *face_classifier_take_3.xml*. Here `faces.vec` is the set of positive samples (sized to width-by-height = 30-by-40), and random images extracted from `backgrounds.idx` will be used as negative samples. The cascade is set to have 20 (`-nstages`) stages, where every stage is trained to have a detection rate (`-minhitrate`) of 0.998 or higher. The false hit rate (`-maxfalsealarm`) has been set at 50% (or lower) each stage to allow for the overall hit rate of 0.998. The weak classifiers are specified in this case as "stumps", which means they can have only one split (`-nsplits`); we could ask for more, and this might improve the results in some cases. For more complicated objects one might use as many as six splits, but mostly you want to keep this smaller and use no more than three splits.

Even on a fast machine, training may take several hours to a day, depending on the size of the data set. The training procedure must test approximately 100,000 features within the training window over all positive and negative samples. This search is parallelizable and can take advantage of multicore machines (using OpenMP via the Intel Compiler). This parallel version is the one shipped with OpenCV.

Other Machine Learning Algorithms

We now have a good feel for how the ML library in OpenCV works. It is designed so that new algorithms and techniques can be implemented and embedded into the library easily. In time, it is expected that more new algorithms will appear. This section looks briefly at four machine learning routines that have recently been added to OpenCV. Each implements a well-known learning technique, by which we mean that a substantial body of literature exists on each of these methods in books, published papers, and on the Internet. For more detailed information you should consult the literature and also refer to the *…/opencv/docs/ref/opencvref_ml.htm* manual.

Expectation Maximization

Expectation maximization (EM) is another popular clustering technique. OpenCV supports EM only with Gaussian mixtures, but the technique itself is much more general. It involves multiple iterations of taking the most likely (average or "expected") guess given your current model and then adjusting that model to maximize its chances of being right. In OpenCV, the EM algorithm is implemented in the `CvEM{}` class and simply involves fitting a mixture of Gaussians to the data. Because the user provides the number of Gaussians to fit, the algorithm is similar to K-means.

K-Nearest Neighbors

One of the simplest classification techniques is *K-nearest neighbors* (KNN), which merely stores all the training data points. When you want to classify a new point, look up its *K* nearest points (for *K* an integer number) and then label the new point according to which set contains the majority of its *K* neighbors. This algorithm is implemented in the `CvKNearest{}` class in OpenCV. The KNN classification technique can be very effective, but it requires that you store the entire training set; hence it can use a lot of

memory and become quite slow. People often cluster the training set to reduce its size before using this method. Readers interested in how dynamically adaptive nearest neighbor type techniques might be used in the brain (and in machine learning) can see Grossberg [Grossberg87] or a more recent summary of advances in Carpenter and Grossberg [Carpenter03].

Multilayer Perceptron

The *multilayer perceptron* (MLP; also known as *back-propagation*) is a neural network that still ranks among the top-performing classifiers, especially for text recognition. It can be rather slow in training because it uses gradient descent to minimize error by adjusting weighted connections between the numerical classification nodes within the layers. In test mode, however, it is quite fast: just a series of dot products followed by a squashing function. In OpenCV it is implemented in the `CvANN_MLP{}` class, and its use is documented in the *…/opencv/samples/c/letter_recog.cpp* file. Interested readers will find details on using MLP effectively for text and object recognition in LeCun, Bottou, Bengio, and Haffner [LeCun98a]. Implementation and tuning details are given in LeCun, Bottou, and Muller [LeCun98b]. New work on brainlike hierarchical networks that propagate probabilities can be found in Hinton, Osindero, and Teh [Hinton06].

Support Vector Machine

With lots of data, boosting or random trees are usually the best-performing classifiers. But when your data set is limited, the *support vector machine* (SVM) often works best. This *N*-class algorithm works by projecting the data into a higher-dimensional space (creating new dimensions out of combinations of the features) and then finding the optimal linear separator between the classes. In the original space of the raw input data, this high-dimensional linear classifier can become quite nonlinear. Hence we can use linear classification techniques based on *maximal between-class separation* to produce nonlinear classifiers that in some sense optimally separate classes in the data. With enough additional dimensions, you can almost always perfectly separate data classes. This technique is implemented in the `CvSVM{}` class in OpenCV's ML library.

These tools are closely tied to many computer vision algorithms that range from finding feature points via trained classification to tracking to segmenting scenes and also include the more straightforward tasks of classifying objects and clustering image data.

Exercises

1. Consider trying to learn the next stock price from several past stock prices. Suppose you have 20 years of daily stock data. Discuss the effects of various ways of turning your data into training and testing data sets. What are the advantages and disadvantages of the following approaches?
    a. Take the even-numbered points as your training set and the odd-numbered points as your test set.